The CRAFT Fortran Programming Model

DOUGLAS M. PASE, TOM MACDONALD, AND ANDREW MELTZER
Cray Research, Inc., 655F Lone Oak Drive, Eagan, MN 55121

ABSTRACT

Many programming models for massively parallel machines exist, and each has its advantages and disadvantages. In this article we present a programming model that combines features from other programming models that (1) can be efficiently implemented on present and future Cray Research massively parallel processor (MPP) systems and (2) are useful in constructing highly parallel programs. The model supports several styles of programming: message-passing, data parallel, global address (shared data), and work-sharing. These styles may be combined within the same program. The model includes features that allow a user to define a program in terms of the behavior of the system as a whole, where the behavior of individual tasks is implicit from this systemic definition. (In general, features marked as *shared* are designed to support this perspective.) It also supports an opposite perspective, where a program may be defined in terms of the behaviors of individual tasks, and a program is implicitly the sum of the behaviors of all tasks. (Features marked as private are designed to support this perspective.) Users can exploit any combination of either set of features without ambiguity and thus are free to define a program from whatever perspective is most appropriate to the problem at hand.

1 INTRODUCTION

The CRAFT programming model is an attempt to allow the user a range of control over the CRAY T3D hardware. This range extends from a low level of control in which the programmer makes almost all of the decisions about how data and work are partitioned and distributed to a high level of control where the programmer identifies where parallelism is located and lets the system determine best how to exploit it. The programming model also allows users to write programs that execute in a data-parallel fashion. It also allows the user to control processor element (PE) execution more explicitly, as occurs in a single-program multiple data (SPMD) model. Thus, one can write a program that specifies what the system as a whole will compute, what each individual task will compute (and the whole program is the sum of the behaviors of all of the tasks), or one that combines elements of both.

The major elements of this programming model include the access to and placement of data, parallel and local execution, work-sharing, synchronization primitives, private and global I/O, subroutine interfaces, and special intrinsic functions that support parallel reductions, parallel prefix operations, and segmented scan operations. The parallel virtual machine (PVM) [1] library is provided to support message-passing programs, but there are no restrictions against using PVM in combination with distributed memory or work-sharing features. Additional directives and intrinsic functions allow the user to access low level

Received September 1993
Revised May 1994

Scientific Programming, Vol. 3, pp. 227–253 (1994)
CCC 1058-9244/94/030227-27

detail about array distributions. The data distribution declarations were adapted in part from Rice University's Fortran D project [2] and Vienna Fortran [3]. The work distribution directives are adapted from a variety of sources, including Cray Autotasking® [4, 5], and workshops on parallel programming held at the University of Illinois at Urbana-Champaign. Concepts embraced in this model can be found in many other sources as well [3, 6–8].

CRAFT was originally conceived before the High-Performance Fortran (HPF) Forum began in 1992 but the two languages are similar, primarily because of the strong influence Fortran D had on both definitions. CRAFT and HPF [9] are based on the belief that current massively parallel architectures attain their highest speeds when the data accessed exhibit high locality of reference. Both languages use data distribution directives to achieve this locality. HPF, however, is a data parallel language. CRAFT supports data parallel programming styles as well but also supports SPMD and message-passing paradigms.

HPF has a much richer set of data distribution capabilities than does CRAFT, but CRAFT allows explicit control of work-sharing and limits the data distribution to those distributions that are deemed to be high performance. On high-bandwidth, low-latency architectures (which is, in our opinion, the future of massively parallel architectures) the cost of determining the location of data based on the possible data distributions with HPF can overwhelm the cost of referencing even nonlocal data. Because CRAFT is SPMD, it also provides the user with data explicitly private to a processing element.

Programs initially execute in parallel. Sequential regions are explicitly inserted by the user. Because of the number of PEs involved, users should try to arrange for minimal sequential regions (e.g., for data initialization). Long sequential segments anywhere within a program could result in highly inefficient machine utilization. A task is not created dynamically in the sense that UNIX® processes are, but rather one task is created per PE at program startup time and parked during sequential execution. No additional tasks can be created dynamically. Each task also has an identity that it can use to distinguish itself from other tasks. The number of executing tasks is available through a special symbolic constant.

This programming model distinguishes between data objects that are private to a task (PE_PRIVATE) and those that are shared among all tasks (SHARED). Private data objects in this model, whether scalars or arrays, are not accessible to any other task. They are not distributed across PEs, but instead each private object is replicated and a copy resides on each PE. Thus, each task that references a private object references its own private version of that object; the storage for the object is replicated across the PEs. It is possible for private data objects associated with different PEs to have different values. These different values can never conflict because there is never an effort to merge different values into a single result. Shared data objects, in contrast, are accessible to all tasks, are not replicated, and (if the object is an array) may be distributed across multiple PEs.

Loops do not create parallelism in a program. Rather they may execute serially, each loop by each task, or they may "share" the work, distributing iterations across all available tasks. Distributed loops (called shared loops in this model) are work-sharing constructs rather than task-creating constructs. Each task is assigned a set of iterations of a shared loop to execute.

Local loops, or loops that are not distributed (called private loops in this model), are included as well. They allow a user to write programs by defining what each individual task will accomplish within each loop. Their behavior individually is most like loops as defined in Fortran 77, i.e., induction and loop control variables behave as they do in Fortran 77, and iteration execution is guaranteed to retain the same ordering as in Fortran 77. These properties of local loops are not shared by distributed loops.

The standard shared memory synchronization primitives are supported in this model. A user can place barriers, locks, critical sections, and events within a program. The implementation of barriers, locks, and critical sections is very efficient, primarily due to extensive hardware support. For example, the current implementation of the barrier operation allows the user to synchronize all PEs (up to the full machine) in approximately 1.5 microseconds.

Subroutine interfaces are extended to accommodate distributed data. Although it is tempting to require that the distribution attributes of actual arguments in function calls exactly match the corresponding dummy arguments in function definitions, it is perceived that such a restriction causes undue hardship on the programmer in many circumstances. On the other hand, supporting such a restriction holds the potential of producing functions that execute significantly faster than their

more general counterparts. This model offers a compromise by allowing a user to specify or not specify the distribution attributes of dummy arguments. When the attributes are given the compiler generates the more efficient code for those references. When they are not, the more general (and less efficient) code is generated. When calls are made to subroutines that have different distributions specified for dummy arguments than the actual arguments they were given, redistribution is done automatically by the compiler.

Although the exact difference in efficiency will depend on the particular code being executed, the cost of array redistribution and shared array addressing deserve attention. Both costs are new to many programmers, and experience in programming uniform memory access parallel machines does not give an accurate intuition about their impact. Significantly more work, in the form of integer computations, must take place to compute the address of a shared array reference. When the array size and distribution are specified, the compiler can "fold" much of that computation so it does not need to be done at run-time. The compiler may also use the folded constants directly in the load instructions, thereby avoiding several trips to memory that would otherwise have to be done if the information were not supplied. Array redistribution, even when done efficiently, can incur a noticeable cost because of the massive amount of data motion involved. The CRAY T3D global network is very fast, but a program's performance can easily be overwhelmed if one is incautious about redistributing arrays.

Two categories of intrinsic functions are supported in this model: high level array syntax functions and low level functions that give information about array distributions, task identity, and whether execution is currently parallel, sequential, or work-sharing. The high level intrinsics operate on entire arrays. The low level intrinsics provide usable information about how an array is distributed across the machine, or what execution region the program is in (parallel, sequential, or work-sharing).

The model supports message-passing primitives based on the PVM model. PVM is a public domain set of portable message-passing primitives originally from the Oak Ridge National Lab. These primitives allow an explicit message-passing style of programming.

Directives were chosen to increase the likelihood that codes written using this programming model will run correctly on machines that do not support the directives. Code written using this model produces mathematically identical results (module hardware limitation considerations) on a sequential machine if the directives are ignored so long as there is no nondeterministic behavior in the user's program and the program does not use any of the machine-specific intrinsic functions.

2 DATA OBJECTS

A data object is any program data storage area, whether it is dynamically allocated, a common block, an array, or a scalar variable. This programming model supports two basic sets of data object attributes above and beyond those allowed by Fortran (specifically `cf77` Version 6.0 [4], Cray Research's Fortran compiler). The first set of attributes is called private because data objects with this set of attributes are private to every processing element; they are accessible only to the task that owns them. The second is called shared because such objects are accessible to all tasks.

2.1 Private Objects

Private data objects are replicated. Each declaration of a private object causes one such object with the specified name to be created for each task that executes. Dynamic private objects are allocated on the private heap. Private data is not distributed. A private object is always allocated entirely within the task and PE that is able to reference it. Private data objects are intended to support, among other things, a user's ability to control the execution of individual tasks at an arbitrarily fine level of detail.

The default distribution attribute is PE_PRIVATE, meaning all data objects are assumed to be private unless explicitly stated otherwise. Variables and arrays may also be explicitly declared as PE_PRIVATE with the directive

```
CDIR$ PE_PRIVATE var1, var2, ..., varn
```

Initial values for private data objects are undefined, but private objects may be explicitly initialized by DATA statements when it is permitted in `cf77` to do so. If it is permitted in `cf77` to initialize a variable with a DATA statement in a sequential program, it is permitted to initialize the same variable declared with the PE_PRIVATE attribute in this model. This means all private data objects may be DATA initialized except those that occur in

blank or unnamed common dummy arguments, automatic arrays, and those whose size is a function of N$PES, the number of PEs executing the program. (N$PES is described in Section 3.2.)

All data objects whose size is a function of N$PES have storage association and sequence association restrictions. An EQUIVALENCE statement cannot specify one of these objects. This includes common blocks. If any entity within the common block is shared or has a size that is a function of N$PES, the entire common block is affected. Essentially, each entity in the original common block behaves as if it were its own separate and unique common block. Thus, storage association and sequence association for the original common block are lost. No entity in the original common block can appear in an EQUIVALENCE statement.

2.2 Shared Objects

Shared data objects are accessible to all tasks. Only one data object exists for each declaration of a shared data object. Blank common blocks may not be shared, nor may objects in blank common be shared, but objects local to a subroutine, including automatic arrays (which may be allocated on the stack), can be shared. Character data may not be shared. A shared object is considered to be distributed across the program's PEs.

The distributions of shared data objects fall generally into two categories: shared scalars and dimensional distributions. Scalar variables are always allocated on a single PE, although not all shared scalars are necessarily allocated on the same PE. Dimensional distributions may be applied to any shared array. They may not be applied to common blocks, although shared arrays in named common blocks can be dimensionally distributed.

With dimensional distributions, each array dimension is distributed as if it were independent of all other dimensions. For this to occur the number of available processors is factored and each array dimension is assigned some factor appropriate to the dimension size and distribution. Thus a three-dimension array mapped to a 64-processor machine might have four processors mapped to each dimension. (This works because $4^3 = 64$.) Alternatively it might have eight processors mapped to the first dimension, four mapped to the second, and two to the third. (Again, $8 \times 4 \times 2 = 64$.) The user may specify a preference for one factorization over another by assigning weights (defined later in this section) to each of the dimensions. The first factorization would be chosen if all dimensions were given the same weight. The second factorization is chosen if the first dimension weight is 4 (because it is 4 times larger than the last dimension), the second dimension weight is 2, and the last dimension weight is 1.

Dimension indexes are mapped to the processors according to the distribution designation specified by the user. Allowable designations are :BLOCK, :BLOCK(M) and ":". (The ":" is required for the directive but will be omitted in the following discussion of the directive.) The first designation, BLOCK, specifies that the dimension is to be divided in such a way that each PE receives one contiguous block of elements. The second designation, BLOCK(M), indicates that each PE is to receive M contiguous elements starting on PE 0. Excess elements are allocated in the same way, again beginning with PE 0. Thus, if a dimension is allocated on 4 PEs using the BLOCK(1) designation, the first element would go to PE 0, the second to PE 1, the third to PE 2, the fourth to PE 3, the fifth to PE 0, and so forth; this distribution is often called cyclic. M may be a constant integer expression or a dummy argument. The last designation, ":", indicates that all elements in the dimension are to be allocated on the same processor. This is often called the degenerate distribution and it implies that the number of processors assigned to that dimension is identically one.

The home processor of a given element can be determined by considering the virtual processor array implied by the array dimension weights and the dimension distributions. These values are linearized in the same way that Fortran linearizes a tuple of array indices to obtain the address of an array element. For example, consider the array declaration REAL X(64). Suppose that this array is to be distributed over four processors. If the array is distributed BLOCK, then 64/4 = 16 array elements in a contiguous block of indices are allocated to each PE. Thus X(1 : 16) is assigned to PE 0, X(17 : 32) is assigned to PE 1, etc. As a second example, suppose that an array Y is declared REAL Y(16, 16) and that it is mapped to an eight PE machine. Furthermore, suppose that the first dimension is assigned four PEs, the second dimension is assigned two, and both dimensions are declared with a BLOCK distribution. The block size for dimension 1 is 16/4 = 4, for dimension 2 it is 16/2 = 8. An arbitrary reference Y(3, 15) maps to a processor tuple of ((3-1)/4, (15–1)/8) which is (0, 1). This tuple and the num-

bers of PEs assigned to each dimension are used to determine the exact PE number. The PE number is given by $0 + 1 \times 4$, where 0 and 1 are from the tuple, and 4 is the number of PEs in the first dimension. A picture of the allocation is given in Figure 1.

Individually distributed arrays always have their first element on PE 0, regardless of the number of elements or the particular distribution used. This allows a user to align distributed arrays in a primitive but efficient fashion to guarantee that when array references are local to one PE that similar references to an aligned array are also local. Note that this only works when the arrays are conformable (they have the same rank and dimension extents) and when they are given the same distribution. For example, consider four arrays A(M, N), B(M, N), C(M, N), and D(K, L), where K, L, M, and N are all distinct values, that are distributed A(:BLOCK, :BLOCK), B(:BLOCK, :BLOCK), C(:BLOCK, :), and D(:BLOCK, :BLOCK). All arrays have the same rank, i.e., they all have two dimensions. Arrays A and B are aligned because they have the same number of elements in each dimension, and each dimension has the same distribution. Array C is not aligned with either A or B because although it has the same number of elements in each dimension, one dimension does not have the same distribution as the corresponding dimension in A or B. Array D has the same distribution for each dimension as A and B, but it has a different number of elements, and thus the elements of D are not all aligned with the elements of A or B. If arrays differ as to the number of dimensions, or if the weight on a distribution designation is not the same, their elements may also not be aligned.

PE 0 (0,0) Y(1:4,1:8)	PE 4 (0,1) Y(1:4,9:16)
PE 1 (1,0) Y(5:8,1:8)	PE 5 (1,1) Y(5:8, 9:16)
PE 2 (2,0) Y(9:12,1:8)	PE 6 (2,1) Y(9:12,9:16)
PE 3 (3,0) Y(13:16,1:8)	PE 7 (3,1) Y(13:16,9:16)

FIGURE 1 Memory allocation pattern for Y, distributed (4:BLOCK, 2:BLOCK).

The array alignment requirements impose some fairly strong constraints on shared array memory allocation. In effect, memory is allocated in stripes across the available processors. If a shared array requires 32 words to be allocated on PE 0, then it requires 32 words on every processor even though the number of elements in the array may be fewer than 32 times the number of available processors. Because the dimension sizes have a multiplicative effect on the total memory allocation, the amount of memory that would be wasted is the memory wasted in the first dimension times the memory wasted in the second, and so forth. If one is not careful, this can represent a lot of memory.

So far the definitions and descriptions provided in this model could apply to block sizes, dimension lengths, and PE allocations with arbitrary integer values. Unfortunately, there is a cost to providing such generality. Briefly described, generating addresses of such arrays with arbitrary dimension sizes from index tuples is expensive. It requires numerous integer operations, including several integer divides, multiplications, and additions. To reduce the cost of generating an address from an index tuple the model currently requires each dimension to be a power of 2; it also requires block sizes and the number of processors assigned to each dimension to be a power of 2.

There is an exception to the rule that each dimension of a shared array be a power of 2. When the last (right-most) dimension has a degenerate distribution, its size need not be a power of 2. Even so, requiring powers-of-2 dimension and distribution sizes is a severe restriction in some cases, and as efficient means are found to support arbitrary array distributions, these restrictions will be lifted. We recognize that these are significant restrictions and efforts are in progress to reduce them. In future these restrictions might be relaxed through allowing all right-most degenerate dimensions to be arbitrary integers, or implicitly rounding the dimension and distribution sizes up to a power of 2, or by finding the means to reduce the cost to a reasonable level where the access is regular, e.g., within loops.

To distribute an array by dimension (only arrays can be distributed by dimension) one need only append a description of each dimension's distribution, with any desired weights, to the array name. This generally has the appearance $\mathrm{var}_i\ (\alpha_1, \alpha_2, \ldots, \alpha_r)$ where r is the rank of the array. The notation αj represents a keyword from the selection :BLOCK or :BLOCK(M), option-

ally with weights, or ":". Weighted dimensions are represented by w:BLOCK or w:BLOCK(M), where w and M are integer expressions. The weights specify a ratio for how many PEs are assigned to each dimension.

When distributing an array that is statically allocated as defined by cf77 (e.g., an entity in a common block or specified on a SAVE statement), w and M must be constant integer expressions, otherwise w and M may be arbitrary integer expressions involving dummy arguments. A distribution of ":" cannot be weighted and means that all elements within the dimension reside on the same processor. Effectively, it is a dimension to which one PE is assigned.

Dimensionally distributed array declarations look like the following example. Note that in this example, the last defined dimension of the array is not a power of 2. This is permitted in this case because the last declared dimension has a degenerate distribution.

```
      REAL A(1024, 1024, 5)
CDIR$ SHARED A(:BLOCK, 4:BLOCK(16), : )
```

Initial values for shared data objects are undefined, but, with the exception of arrays whose size is a function of the N$PES constant, shared objects may be explicitly initialized by DATA statements where it is permitted in cf77 to do so. If it is permitted in cf77 to initialize a variable with a DATA statement in a sequential program, it is permitted to initialize the same variable declared with the shared attributes in this model. Primarily this means shared objects may be DATA initialized except those that occur in blank or unnamed COMMON, dummy arguments, automatic arrays, and those whose size is a function of N$PES.

It is sometimes valuable to assert to the compiler that all accesses to any array within a program unit will be resident on the accessing PE. This may speed up access time and allow the data to be cached. A directive has been provided to make this assertion. Its syntax is

```
      REAL X(1024)   !X is a dummy argument
CDIR$ PE_RESIDENT X
```

The directive PE_RESIDENT assetts that all accesses to dummy argument X will be to those elements of X, which are on the accessing PE. Undefined behavior will result if the assertion is not adhered to by the user. The assertion is only allowed on shared dummy arguments.

2.3 Geometry

The concept of geometry in this model is an abstraction of the dimensional distribution. It simplifies the maintenance and declaration of arrays with similar dimensional distributions. One can think of it as providing a shorthand for declaring dimensionally distributed arrays. This is similar to the typedef declaration in C.

The syntax for declaring a geometry name is similar to the syntax used to declare dimensionally distributed arrays. The syntax for declaring a distributed array from a geometry is similar to the Fortran 90 syntax for declaring variables with derived types.

```
CDIR$ GEOMETRY geom(α1, α2, ..., αr), ...
CDIR$ SHARED (geom) [::] var1, ..., varn
```

Here, α_i has the same meaning as it has in Section 2.2 and [::] indicates that the two colons are optional. A user would not actually type the square brackets. The following example demonstrates how to declare a geometry.

```
CDIR$ GEOMETRY G(1:BLOCK, 2:BLOCK)
      REAL A(4, 8), B(16, 8)
CDIR$ SHARED (G) :: A, B
```

The declaration of G describes a distribution that is then applied to arrays A and B. Figure 2 shows the distribution of A across eight PEs.

2.4 Array Redistribution

Some applications can efficiently execute with all data being only statically distributed, but not all applications are like that. It is sometimes the case that a given data layout may yield efficient execution for some phase of the computation, but yield poor efficiency for some other part of the computation. If the two sections of code have sufficient work in them it might be desirable to redistribute the arrays dynamically to maximize reference lo-

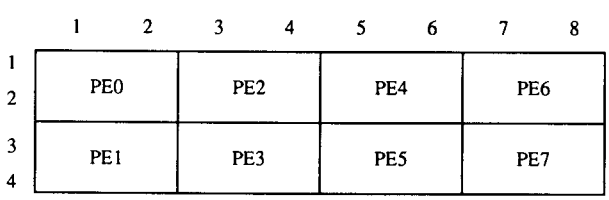

FIGURE 2 Memory distribution of A.

cality. This can be done by declaring additional arrays with the desired distributions, then copying data into the appropriate array just before executing the section of code in question. This does have the disadvantage of increasing memory usage.

Arrays may also be implicitly redistributed across subroutine boundaries (implicit redistribution). A dummy argument that is distributed differently than its actual argument in the calling routine is automatically redistributed upon entry to the subroutine by run-time libraries and automatically redistributed to its original distribution at the subroutine exit. If the distributions are identical or the UNKNOWN or UNKNOWN_SHARED directive is used (Section 7.1.1), no redistribution occurs. All tasks must participate in implicit array redistributions.

2.5 Storage Association and Sequence Association

There are certain guarantees made by `cf77` about the layout of data objects in memory. This layout is defined in terms of storage association and sequence association in the Fortran 77 standard. For example, sequence association semantics define the behavior of using a one-dimensional array to reference elements of a two-dimensional array, and referencing two adjacent arrays in the same common block. Rules of storage association govern program behavior when two arrays are associated in an EQUIVALENCE statement. Distributed data objects and data objects whose size is a function of N$PES do not have the same guarantees made by the Fortran 77 storage association and sequence association semantics. Because these are new kinds of data objects, additional restrictions are placed on the use of EQUIVALENCE statements, argument passing, and common blocks.

Issues with sequence association appear in two places in particular: association between successive dimensions within arrays and association between objects within COMMON blocks. In CRAFT PE-private arrays, as in Fortran, for the fastest running (left-most) dimension, element i is stored immediately following element $i - 1$ and immediately before element $i + 1$. This is true whether or not elements $i - 1$, i, and $i + 1$ are within the declared extents of that dimension, as long as they are not outside the storage allocated for the array. This fact is exploited when a program allocates an array with one set of extents, then passes it to a subroutine and declares it to have a different set of extents. For example, a code may manipulate a work array as a two-dimensional array, then hand the whole array to another routine where it is manipulated as a one-dimensional array. This type of reshaping does not work with shared arrays because elements $i - 1$, i, and $i + 1$ are not necessarily contiguous. They may not even be on the same processor. By the same token, the element that follows i in a given PE's storage may not be $i + 1$. Figure 3 illustrates this issue.

Objects whose size is a function of N$PES have similar problems with sequence association and are similarly restricted. A similar but less complicated problem occurs with objects in COMMON storage. If two objects, say arrays A and B, are lexically adjacent in their declaration within a Fortran program, they are allocated as adjacent storage within the executable. Thus, one element past the last element of A is the first element of B. This is sometimes exploited to give efficient memory management of scratch arrays. This type of storage association cannot be guaranteed when shared arrays are allocated in COMMON blocks. Padding between shared objects may be needed to maintain other properties and the last element of one array may be allocated on a different PE than the first element of the next array.

Shared arrays may not be associated in an EQUIVALENCE statement because elements of the arrays would not have a mapping between them that is remotely similar to that which is provided in Fortran 77. For example, if two arrays `A(64, 128)` and `B(128, 64)` are associated, Fortran requires that array cells `A(I, 2*J-1)` and `B(I, J)` map to the same memory location and references `A(I, 2*J))` and `B(I+64, J)` would also map. If they were distributed `A(:BLOCK(M1), :BLOCK(M2))` and `B(:BLOCK(M1), :BLOCK(M2))`, the same association would hold as for Fortran. However, if the block sizes or num-

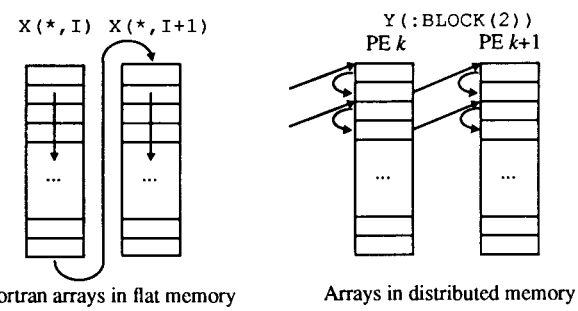

FIGURE 3 Private and shared array sequence association.

ber of PEs allocated to a dimension are different, or the number of dimensions are different, the storage association would not be the same as defined by Fortran. Although it is conceivable that EQUIVALENCE could be permitted for arrays that are appropriately distributed, its usage would be highly error prone, occasionally expensive to verify, and generally not worth the effort required to reasonably support. For this reason shared arrays may not be associated in EQUIVALENCE statements.

2.6 Shared to Private Coercion

When a shared array is passed into a routine that declares the argument as PE_PRIVATE, in effect only those values that reside on the current PE are being passed. This means that the callee must declare each dimension of the array to be as large as the number of elements resident on that PE. For example:

```
      REAL A(256,64)
CDIR$ SHARED A(:BLOCK,:)
```

If this array were passed to a routine that declared the dummy argument as PE_PRIVATE, each PE would see an array of the shape:

```
REAL A(256/N$PES,64)
```

(N$PES and MY_PE are described in Section 3.2.)

In general, the caller is responsible for calculating the local extent of each dimension and passing it to the callee. Sample code for this is as follows (HIIDX, LOWIDX, and BLKCT intrinsic functions are described in detail in Section 8.2):

```
C   Calculate the local allocation of each
C   dimension.
C   This should be done for each dimension.
C   Degenerate dimensions are not reduced
C   in size.
    IBLKSIZE = HIIDX(A,1,0,1) -
         * LOWIDX(A,1,0,1)+1
    N1 = BLKCT(A,1,MY_PE()) * IBLKSIZE
```

There will be cases when a PE has no allocation. If this occurs, the caller or callee must ensure that no work is done on that array for PEs that have no allocation. This will occur in the above example if N$PES is greater than 256. In this case, N1 would be zero for some executing tasks.

If an element of an array is passed instead of passing the entire array, the caller should ensure that the element is local. If the element is local, then the first element passed to each PE will be the element that is specified in the caller. For example:

```
      REAL A(256,64)
CDIR$ SHARED A(:BLOCK(2),:)
      IBLKSIZE = HIIDX(A,1,0 1) -
           *. LOWIDX(A,1,0,1)+1
      N1 + BLKCT(A,1,MY_PE()) * IBLKSIZE
      N2 = 64
CDIR$ DOSHARED (I) ON A(I,1)
      DO I=1,256
         CALL COERCE1(A(I),N1,N2)
      END DO
```

In this case each task gets elements that are local, because the "ON A(I,1)" clause of the loop (see Section 4.1) ensures that the owner of A(I,1) is the PE that executes iteration I. What actually occurs when the call is executed is that the SHARED address in the argument list is changed into a PE_PRIVATE address. This is done by calculating the offset of A(I,1) on its own home PE and using that as a local address. In the above example that is exactly what is expected. If a nonlocal element is passed, the callee on each PE will get an element that corresponds to that offset. For example:

```
CDIR$ SHARED B(:BLOCK(2))
C     Pass the first element. Each PE gets offset 0 in their local
C     allocation of the array. This is the same as CALL COERCE2(A,N1)
      CALL COERCE2(B(1),N1)
C     Pass the third element. This resides at offset 0
C     on PE 1. Each PE gets offset 0 in their local
C     allocation, so it will be exactly the same as above.
      CALL COERCE2(B(3),N1)
C     Pass the second element. This resides at offset 1 on PE 0.
```

```
C     Each PE gets offset 1 in their local allocation.
C     PE 0 gets B(2), PE 1 gets B(4), PE 2 gets B(6)
      CALL COERCE2(B(2),N1)
C     ...
      SUBROUTINE COERCE2(X,N)
CDIR$ PE_PRIVATE X
      DIMENSION X(N)
```

3 TASK EXECUTION

The CRAFT programming model supports the notion of work-sharing on shared data. Constructs within this model provide access to mechanisms that distribute work among the available executing tasks. Shared data are distributed across PEs independently of executing tasks. The model supports both sequential regions (code segments executed by a single task) and parallel regions (code segments executed concurrently by one or more tasks). To simplify programming for some situations, each task is given a unique name. The name is an integer value between 0 and N$PES-1, inclusive.

A program begins execution with all tasks running. Each task is able to execute independently until it reaches a synchronization point, at which time it waits until the synchronization conflict is cleared. Tasks can agree to cooperate by sending messages back and forth through explicit message-passing, by synchronized access to distributed shared memory, or by entering a work-sharing construct. The program retains control over each task until the program terminates. Task creation and scheduling are quite different from a "traditional" fork–join model where tasks are created and destroyed dynamically, and the number of processors available or in use can vary from instant to instant. They are similar in some regards to gang scheduled machines and traditional message-passing machines.

This approach to task execution was chosen for several reasons. This programming model is designed to exploit the power of a distributed memory machine with many processors. The fact that remote references are relatively expensive compared with local references causes processor coordination and synchronization efforts to be relatively expensive when compared with what is required to accomplish the same thing on a uniform memory access machine. Also, the number of processors is large enough that the cost of even small synchronization delays is fairly expensive because of the compute capacity that is lost across the many PEs that become idle waiting for task coordination to complete. A third reason is that this approach is simple for users to understand and use to their advantage. In all it appeared that a mechanism that provided tasks on demand, along with supporting mechanisms for swapping in other jobs to utilize any unused processors, would have strongly interfered with the programmer's ability to control data distribution. It also would have been hard to understand, complex to implement, slow, and would use up a lot of memory in system functions that would otherwise be available to the user. A design that offered speed, simplicity, and low memory use seemed more desirable.

From within code executing in parallel the user may execute the STOP or ABORT statements. The STOP statement stops only the PE executing the statement; all other PEs remain executing and deadlock may occur if the stopped PE is required at a synchronization point. The ABORT statement forces all PEs within the job to cease executing, although all may not cease execution at the same moment.

3.1 Sequential Regions

Programs initially begin executing in parallel. The program remains in that execution mode until it encounters a special directive to execute sequentially. The syntax for this directive is:

```
CDIR$ MASTER
```

The program then continues to execute as a single sequential task until the directive:

```
CDIR$ END MASTER
```

is encountered. Every function or subroutine that contains a MASTER directive must also contain a properly nested matching END MASTER directive. MASTER directives carry an implicit barrier syn-

chronization. The END MASTER directive may also optionally contain a COPY(var$_1$, . . .) clause. This directs the contents of the private data for *var*$_1$, etc., that is owned by the master task to be broadcast to all of the other PEs, basically ensuring that they all start with identical values for each copied data item. An assumed size array cannot appear in the COPY list of an END MASTER. The syntax for an END MASTER with a COPY is:

```
CDIR$ END MASTER, COPY(var1, var2, ...)
```

All tasks are created on program startup and each is attached to a specific PE. Only one task is created per PE. The task on PE 0 is special in the sense that when the program executes in a sequential region (unless it is within a shared loop), it is the master task that executes. While executing in a sequential region all other tasks are parked at the matching END MASTER directive. This mechanism allows tasks to be parked and unparked rapidly. However, because of the number of processors involved, programs containing sequential sections with lots of work will inherently execute well below machine peak performance rates regardless of how rapidly tasks can be parked and unparked.

Sequential regions may be nested but the effect is that the inner directive is ignored. If a subroutine call takes place within a sequential region, the subroutine will execute sequentially; there is no way to "get back" into a parallel region within the subroutine. If the subroutine call takes place within a parallel region a sequential region may be entered from anywhere within the subroutine. Note that encountering an END MASTER directive does not guarantee that the program will resume executing in parallel unless it is the outermost END MASTER directive.

An intrinsic is provided that allows the user to query whether the code is currently executing in a parallel section. This intrinsic is called IN_PARALLEL(). It value is .FALSE. if the code is executing within a sequential region, .TRUE. otherwise. The IN_PARALLEL function must appear on an INTRINSIC statement before use. When the STOP statement executes within a sequential region, tasks parked at the END MASTER directive are also stopped.

3.2 Task Identity

At times it is useful to make decisions within a program based on the number of processors available, both at compile time and at run-time. To allow this the model supports a special symbolic constant called N$PES, which gives the total number of central processing units (CPUs; as well as the number of tasks) available to the program. It may be used in some places a named constant must appear. It cannot be used in DATA, CHARACTER, or FORMAT statements. It has the same value in either a sequential or parallel region. Furthermore, N$PES can only be an operand of the following operators: + − * /. Any constant expression that contains N$PES as an operand is called a symbolic constant expression.

Each task that executes within a program has a unique identification. The name given to each task is retrieved by using the intrinsic function MY_PE(). This function must appear on the INTRINSIC statement before use. It is an integer value between 0 and N$PES-1, inclusive. It may be used anywhere an intrinsic function may be used. It is available whether the program is executing inside a sequential region or a parallel region. When executing in a sequential region it always returns the value 0 unless executing from within a shared loop.

The logical task topology defined by the intrinsic function MY_PE() defines a one-dimensional torus (or mesh). Wraparound is achieved by treating the operations as operations in a linear congruence, i.e., by doing a "mod N$PES" operation to any manipulation within the space. For example, a task may find one of its logical neighbors in the torus by evaluating the expression (MY_PE()+1) mod N$PES. It should be noted that a nearest logical neighbor is not necessarily a nearest physical neighbor.

4 LOOPS

As with data objects, loops have a notion of shared-ness and private-ness. Private loops are executed in their entirety by the task that invokes them. No work is shared between tasks. Private loops define program behavior by defining the behavior of individual tasks. Private loops are defined, at the task level, as having exactly the same semantics as loops in standard Fortran. Iterations are executed in the Fortran-specified order, which implies that (replicated) induction variables retain the behavior they have in sequential Fortran programs. (Induction variables that are shared, of course, suffer from extensive race conditions because many tasks could be concurrently executing the same loop.) No special syntax is required to specify a loop as private—it is the default. Private

loops may, of course, reference both shared and private data. Shared loops specify the behavior of all tasks collectively and define the behavior of individual tasks only implicitly. They permit work specified in the loop to be shared across all tasks. Shared loops do not guarantee the order in which iterations will be executed. The lack of a defined ordering allows the system to execute iterations concurrently. A loop is shared only if it is executed in a parallel region (otherwise it is private). All tasks must participate in the execution of a shared loop. Shared loops may be written explicitly or by using array assignment syntax.

A shared loop is executed as if there are no cross-processor dependencies. Each iteration is executed atomically on a PE. An iteration of a shared loop executes as if it were a single MASTER region (although it does not necessarily execute on PE 0). No function that allocates new shared data storage may be called from within the iteration. This restriction also applies, to some extent, to any function that requires the cooperation of all PEs because of the uncertainty that all PEs will execute the same number of iterations, or even that all PEs will receive some iterations of a given loop invocation. (Functions may test internally whether they are being called from within a shared loop by using the IN_DOSHARED intrinsic described later.)

4.1 Explicit Shared Loops

Explicit shared loops, also called DOSHARED loops from the keyword in the directive, most closely resemble Fortran DO loops in their structure. Their similarities are obvious. They deviate from DO loops in that the sequencing of iterations is lost and that execution of iterations is permitted to, but not required to, occur concurrently. Thus statements such as `X(I)=S*X(I-1)` can produce significantly different results when executed in a DOSHARED loop than in a Fortran DO loop. The general syntax for shared loops is as follows:

```
CDIR$ DOSHARED (I1, I2,...,In) ON X (f1(Ī),f2(Ī),...,fr(Ī))
      DO I1 = L1, U1, S1      ! Must be tightly nested
         DO I2 = L2, U2, S2
            ...
               DO In = Ln, Un, Sn
                  ...
               END DO
            ...
         END DO
      END DO
```

where `n` is the number of shared loops in the nest and `r` is the rank of the array `X`. All index expressions must be of the form $f_i(\bar{I})=aI+b$, where the loop control variable `I` is used in at most one array index expression and `a` and `b` must be integer values and may be expressions, constants, or variables. Therefore `n` must be less than or equal to `r`. An array that has been declared UNKNOWN cannot be used as the target of an alignment. The order of the variables $I_1, I_2, \ldots, I_n$ in the DOSHARED directive is significant.

Private loops may occur inside or outside of the shared loop but the shared loops themselves must be tightly nested. In the event that a shared loop is nested (but not tightly nested) inside a containing shared loop, the inner shared loop is executed as a private loop.* An imperfectly nested shared loop is executed as a private loop whether it is within the same program unit or it is a shared loop contained within a subroutine that is called from within a shared loop. An intrinsic is provided that allows the user to query whether the code is currently executing in a DOSHARED section. This intrinsic is called `IN_DOSHARED()`. Its value is `.TRUE.` if the code is executing within a DOSHARED section, `.FALSE.` otherwise. The IN_DOSHARED function must appear in an INTRINSIC statement before use.

An example of nested DOSHARED loops is as follows:

```
CDIR$ DOSHARED (I) ON A(I)
      DO I = 1,N
         ...
```

(*continues on next page*)

* We recognize that there are differences in behavior between private and shared loops and that treating as private a loop intended to be shared could cause surprises. Although shared loops carry no guarantee of concurrent execution that private execution would violate, the loop alignment cannot be honored, which could result in subtle problems, especially in performance.

```
CDIR$     DOSHARED (J) ON B(J)
          DO J = 1,M  ! private loop
C         nested DOSHARED ignored
             ...
          ENDDO
          ...
       ENDDO
```

Program correctness depends in part on whether a loop is declared as private or shared (and if shared, whether it executes as a shared or private loop). In most cases private and shared loops have different behaviors. In addition, they function identically in sequential sections.

Proper choice of iteration alignment can often provide a high degree of locality when references in the iteration are close together. The aligned distribution mechanism is designed to place iterations within tasks on PEs where the references reside. For example, suppose that arrays X and Y have the same dimensionality, the same size, and the same distribution. The loop:

```
CDIR$ DOSHARED (I) ON X(I)
      DO I = 1, N
         Y(I) = A*X(I) + Y(I)
      END DO
```

is distributed such that each iteration I is executed on the processor where X(I) resides. Because Y(I) resides on the same PE, all references are completely local.

4.2 Array Assignment

Array syntax is supported in this programming model. Array assignment statements involving shared arrays are treated as if they were shared loops. Unlike shared loops, their iteration distribution is controlled completely by the compiler. The compiler chooses the iteration distribution that exercises the greatest locality in its execution. This may entail distributing operations within a single iteration across multiple tasks, something that a user is not able to do with DOSHARED loops. For example, given the declaration:

```
      DIMENSION A(128), B(128), C(128)
CDIR$ SHARED A(:BLOCK), B(:BLOCK(2))
CDIR$ C(:BLOCK(2))
```

When A is SHARED, the array syntax assignment:

$$A = B + C$$

is semantically equivalent to:

```
CDIR$ DOSHARED (I) mechanism
      DO I=1,128
         A(I) = B(I) + C(I)
      ENDDO
```

where mechanism is chosen by the compiler. When A is PE_PRIVATE it is equivalent to a private loop (even if B and C are shared arrays).

4.3 Barrier Removal

Barriers are implicitly included at the end of every distributed loop including array assignments involving shared arrays, but the compiler is at liberty to remove them when program analysis determines that it is safe to do so. Implicit barriers also exist when data are redistributed and when SHARED automatic arrays are allocated. Barriers can only be explicitly removed from shared loops by placing a directive at the end of the loop. The syntax of that directive is

```
CDIR$ NO BARRIER
```

Barriers associated with MASTER directives, implicit redistribution, and SHARED automatic array allocations cannot be removed.

Barrier removal must be done with great care. The compiler exploits the locality of reference available within a shared loop. Removing the barrier does not necessarily invalidate the caching scheme used for local references. If a barrier removal allows shared data that were referenced locally in a shared loop to be referenced remotely prior to the next synchronization point, then the NO BARRIER directive must be used in conjunction with a

```
CDIR$ SUPPRESS
```

directive to ensure that all data are returned to memory. The SUPPRESS directive has the effect of forcing a PE's local cache to be flushed.

5 SYNCHRONIZATION PRIMITIVES

This model supports a standard array of shared memory synchronization mechanisms, including barriers, locks, critical regions, and events. Each type of synchronization mechanisms is supported by special hardware to make the operation as efficient as possible. Each type of mechanism is described in the following sections.

5.1 Barriers

Barriers are a mechanism for synchronizing all tasks at once. Entering a barrier causes a task to stall until all tasks have entered the barrier. Barriers are expected to be extremely fast—the current implementation takes about 1.5 microseconds. They are implicitly included at the end of every distributed loop, but the compiler is at liberty to remove implicit barriers when it is safe to do so. Barriers are also included at the beginning and end of routines that allocate distributed memory dynamically (discussed in Section 7). They may be explicitly included anywhere in a program with the syntax

```
CDIR$ BARRIER
```

Barrier directives may occur in sequential regions of a program with no effect whatsoever. Conceptually it is one task synchronizing with itself. Permitting barriers in sequential sections allows users to write subroutines that may be executed sequentially or in parallel, and if executed in parallel require synchronization.

The BARRIER routine is equivalent to the BARRIER directive except that it is not ignored inside sequential regions. Its purpose is to provide direct access to the hardware barrier mechanism and as such it requires more caution in its use. For example, if used within a master region it can cause undesirable program behavior. It is also somewhat faster than the BARRIER directive. Its syntax is

```
CALL BARRIER()
```

The barrier mechanism actually consists of two parts, setting the barrier and waiting for the barrier to clear. The point at which a processor sets the barrier and the point at which the processor waits for the barrier need not coincide. They do coincide with the BARRIER directive. However, there are some applications that lose a large amount of time waiting at barriers when the computation preceding the barrier is not homogeneous. The following three routines allow early arriving processors to move forward into an independent phase of the computation while the slower processors catch up:

```
CALL SET_BARRIER()
CALL WAIT_BARRIER()
IB = TEST_BARRIER()
```

SET_BARRIER sets the barrier. It indicates that the calling task has arrived at a barrier synchronization point. WAIT_BARRIER suspends task execution until all tasks arrive at the barrier. TEST_BARRIER returns the state of the barrier: zero if barrier is not satisfied, nonzero otherwise.

The following is an example of barrier functions:

```
C   block 1: must be completed before
C   block 2 is started
    CALL SET_BARRIER();
C   unconstrained calculations
    CALL WAIT_BARRIER();
C   block 2: cannot be started until
C   block 1 is completed
```

It is important to note that the BARRIER(), SET_BARRIER(), and WAIT_BARRIER() routines are not ignored, and therefore must not occur, inside sequential regions (unlike the BARRIER directive).

5.2 Locks

Locks are a basic and primitive synchronization mechanism that are generally used to serialize access to some piece of data. They are basic in the sense that they may be used to efficiently implement a variety of parallel constructs, including other synchronization constructs. They are primitive in the sense that serialization is enforced by convention only. Access to the lock is serialized by a combination of hardware and software, but if a lock is to be used to serialize access to some datum X it is the responsibility of the programmer to ensure that no section of code accesses X without first gaining access to the lock. Locks themselves are only partially protected from unauthorized access in that the operation of locking a lock is serialized, but unlocking a lock is not protected. Only one task may set the lock on, but any task may clear the lock.

Locks do not require initialization or release

functions. Locks that are set to zero are initialized as unlocked. Lock operations are supported by three library subroutines, which use the syntax

```
CALL SET_LOCK (lock)
CALL CLEAR_LOCK (lock)
L = TEST_LOCK (lock)
```

The subroutine SET_LOCK sets the lock. If the lock is set in spin-waits until the lock is cleared, otherwise the lock is set immediately. CLEAR_LOCK clears a lock whether it is set or not. TEST_LOCK atomically sets a lock and returns the state that the lock had (whether set or cleared) prior to the test. With this function a task can avoid blocking on a set lock by testing the lock. If the lock is clear, the testing task will have set the lock. Otherwise, the task will be informed and it will be free to perform some other operation.

For example,

```
IF (.NOT. TEST_LOCK(lock)) THEN
   ...(the lock is ours)...
   CALL CLEAR_LOCK(lock)
ELSE
   ...(do something else)...
END IF
```

For all three functions the operand lock is a shared 64-bit integer.

These queuing-locks are designed to efficiently support many locks that infrequently have access collisions. Contention-free access is inexpensive—clearing a lock with one PE blocked costs 4 microseconds. Access with contention is somewhat more expensive, e.g., clearing a lock with 15 PEs blocked costs approximately 15 microseconds.

5.3 Critical Sections

Critical sections are a specialized form of lock that do not require the use of some convention to ensure proper synchronization. They serialize access to a particular section of code rather than access to some data object. A critical section prevents more than one task from executing concurrently within the critical section. The syntax for a critical section is

```
CDIR$ CRITICAL
CDIR$ END CRITICAL
```

Every CRITICAL directive must have a properly nested and matching END CRITICAL directive within the same routine. The only way to enter a critical section is through the CRITICAL directive (i.e. no branching in) and the only way to exit the critical section is through the END CRITICAL directive.

5.4 Events

Events provide a style of program synchronization that is different from locks. Whereas locks cause task suspension on setting the lock, events have an explicit blocking routine. Events are typically used to record the state of a program's execution and communicate that state to other tasks. Because events have no atomic operation to set a lock and block on conflict, events cannot be as easily used to completely serialize access to data.

This mechanism is supported by four library routines, namely SET_EVENT, WAIT_EVENT, TEST_EVENT, and CLEAR_EVENT. SET_EVENT set, or post, an event. It declares the event to have occurred. No prior conditions are imposed on this routine. Any event can be posted at any time, whether the state of the event is already posted or cleared. WAIT_EVENT suspends task execution until a specified event occurs. TEST_EVENT returns the state of an event, i.e., whether it is posted or cleared. CLEAR_EVENT, of course, clears the event. The syntax for each routine is:

```
CALL SET_EVENT([event])
CALL WAIT_EVENT([event])
CALL CLEAR_EVENT([event])
S = TEST_EVENT([event])
```

The argument to the event routines is optional. If an argument is supplied, it must be a shared integer variable or array element. If these routines are called without an argument, then a fast hardware mechanism is used in place of the somewhat slower, but more versatile, software mechanism. (The hardware mechanism is called the *eureka* mechanism because posting an event is like shouting "Eureka! I found it!") By comparison, eureka events cost 1.5 microseconds whereas software events cost a few tens of microseconds. On the CRAY T3D the eureka mechanism shares the same hardware used by barriers. Barriers are implemented using an AND tree. Each PE writes a 0 to a special register to arm the mechanism; each PE then writes a 1 as it enters the barrier. When all PEs have written a 1, the AND tree inverts its value from a 0 to a 1. For a eureka each PE writes a 1 to arm the mechanism, which sets the AND tree

to 1. When any PE writes a 0 to the register, the AND tree reverts to 0.

5.5 Atomic Update Statements

Vector updates are assignment statements that modify or update an array reference that has an element of indirection. An example of this is

```
X(IX(I)) = X(IX(I)) + V(I)
```

The concern is that IX may contain values that occur more than once. When this is the case, executing the update in parallel can cause race conditions that may result in incorrect values for X.

Vector updates pose a difficult problem. The option of computing vector updates sequentially to avoid potential race conditions is generally not acceptable for performance reasons, and forcing the programmer to do all of the necessary synchronization by hand is error-prone and unnecessary. This programming model supports a set of vector update primitives for floating-point and integer addition, multiplication, maximum, minimum, and for binary AND, OR, and exclusive OR operations. The directive ATOMIC UPDATE directs the compiler to ensure that multiple updates to a single shared element occur atomically. The vector update executes as parallel as possible otherwise. As for all floating-point operations executed in parallel, care must be used to be sure that the order in which floating point ATOMIC UPDATE operations are performed does not contribute to excessive numerical instability in the program. An example of ATOMIC UPDATE usage follows:

```
CDIR$ DOSHARED (I) ON IX(I)
      DO I=1, N
CDIR$    ATOMIC UPDATE
         X(IX(I)) = X(IX(I)) + V(I)
      ENDDO
```

The ATOMIC UPDATE directive only applies to the assignment statement immediately following the directive and may be placed before nonvector updates as long as one of the supported operations is being executed and the assignment is to a shared data object. The statement following the ATOMIC UPDATE directive must be an assignment statement.

Another example of how the ATOMIC UPDATE directive can be used is the following, which computes a sum reduction.

```
CDIR$ SHARED SSUM, A(:BLOCK)
CDIR$ MASTER
      SSUM = 0
CDIR$ ENDMASTER
      PSUM = 0
CDIR$ DOSHARED (I) ON A (I)
      DO I=1, 100
         PSUM = PSUM + A(I)
C           LOCAL SUMS
      END DO
CDIR$ ATOMIC UPDATE
      SSUM = SSUM + PSUM
C       ACCUMULATE LOCAL SUMS
```

5.6 Shared Data Coherence

It is important to understand when shared data objects are coherent. The CRAY T3D supports cache coherency for private data objects, but not for shared objects. Thus, only shared data objects can become incoherent. When coherent data objects are accessed from memory, the most recently updated value is always obtained. Accessing incoherent data objects may result in stale values being obtained. A value becomes stale, e.g., when the most recently computed value is still in a register or in a PE's local cache and not in memory. To further clarify this point, suppose two PEs use a shared value without proper synchronization; one PE reads the value into a register and uses it, another writes it, then the first PE reuses its local (but stale) copy as if it were the most up-to-date value. Using the local (in register) copy is faster than rereading the value from memory, but it is not what the user expected and is likely to give incorrect results.

Shared data objects are forced to become coherent immediately before an external call to a subprogram and after a synchronization point. Shared data objects become incoherent after being modified with a new value. Shared data coherence is only a consideration when one task is modifying a shared data object and a different task is referencing the same object. If the same task is both accessing and modifying the same shared data object, shared data coherence is irrelevant.

The following is a list of points after which data are guaranteed to be coherent:

1. Implicit barrier synchronization points
2. Barrier synchronization points specified by the BARRIER directive
3. The BARRIER function
4. The WAIT_BARRIER function

5. The TEST_BARRIER function
6. The SET_LOCK function
7. The TEST_LOCK function
8. Critical region synchronization points specified by the CRITICAL directive
9. The WAIT_EVENT function
10. The TEST_EVENT function
11. The ATOMIC UPDATE directive

Even though data become coherent, it is still necessary to control communication involving those data with appropriate synchronization. The following example shows that coherent shared data may still cause problems if communication is not controlled with some synchronization mechanism. The shared scalar X is only updated by a single task (PE 13), but no synchronization occurs. Therefore, some of the other tasks may access the old value of X.

```
      COMMON /SCALAR/ X
      DATA X /3.0/
CDIR$ SHARED X
      INTRINSIC MY_PE
      IF (MY_PE() .EQ. 13) X = 5.0
      CALL RTN()
      Y = X ! nondeterministic X
      PRINT *, 'MY_PE = ', MY_PE(),
     *    'Y =', Y
      END
      SUBROUTINE RTN()
      COMMON /SCALAR/ X
CDIR$ SHARED X
      INTRINSIC MY_PE
      Z = X ! nondeterministic X
      PRINT *, 'MY_PE = ', MY_PE(),
     *    ' Z = ', Z
      END
```

The following example shows one way to ensure that X is safe to access:

```
      PROGRAM NORACE
      COMMON /SCALAR/ X
      DATA X /3.0/
CDIR$ SHARED X
      INTRINSIC MY_PE
CDIR$ MASTER ! implicit synchronization
         X = 5.0
CDIR$ END MASTER
      CALL RTN()
      Y = X ! deterministic X
      PRINT *, 'MY_PE =', MY_PE(),
     *    ' Y = ', Y
      END
      SUBROUTINE RTN()
      COMMON /SCALAR/ X
CDIR$ SHARED X
      INTRINSIC MY_PE
      Z = X ! deterministic X
      PRINT *, 'MY_PE = ', MY_PE(),
     *    ' Z = ', Z
      END
```

In the above example, the MASTER directive contains an implicit synchronization point. This causes X to become coherent and communication involving X is appropriately controlled. Thus synchronization and coherence are both necessary for correctness.

6 INPUT AND OUTPUT

Input and output may be accomplished by using either private or global I/O. Private I/O is the default.

6.1 Private I/O

A private READ or WRITE statement is one that, when encountered, is executed in its entirety by the processor that encounters it. It requires no synchronization across, nor communication with, other processors. It is executed without regard for the activity of other processors. Thus, one processor, or a thousand, may execute a READ or WRITE statement concurrently.

Shared and private variables alike may be used in private I/O statements. Each processor has access to all shared data and to its own PE-private data. Of course, all access to shared data must be carefully coordinated across all processors, perhaps using explicit synchronization to avoid read/write and write/write conflicts (race conditions) on the shared memory.

Private OPEN operations, like READ and WRITE operations, are executed by a single processor rather than cooperatively by many processors. A single task that encounters an OPEN statement will execute the OPEN without coordination or communication with other tasks. The file unit opened is private and accessible only to the task that executed the OPEN statement. Any operations performed using the unit will not affect the state of any other task. If the same file is opened by another task, then reads will cause both tasks to read all the data and writes will cause undefined behavior.

Because private I/O is the default, all READ, WRITE, OPEN, CLOSE, and INQUIRE operations

will be private operations unless explicitly declared otherwise. Private I/O is useful when a programmer wishes to specify the I/O behavior from the perspective of what each task does, or when a task must write private data. No other task is required to participate in private I/O so it may be used to achieve unsynchronized I/O as well.

Aside from the aspect of read/write and write/write ordering conflicts across tasks, private I/O is identical to Fortran 77 I/O on a serial machine. It supports all of the various flavors, including direct, sequential, formatted, unformatted, and list-directed operations.

Private I/O statements cannot have shared data allocations, work-sharing, or barriers in functions called from within the I/O statements. Properties of private I/O:

1. If multiple PEs try to read the same file sequentially, then all PEs will read all of the data on the file.
2. If multiple PEs try to write the same file sequentially, the results are undefined (except for shared, specially buffered files such as `stderr`, which is line-buffered).

6.2 Global I/O

Global I/O is similar to private I/O except that the unit/file connection is a global, shared resource. The global I/O paradigm offers two significant benefits over private I/O: clarity and performance. Consider, for example, an embarrassingly parallel application. In such an application the parallelism exists at the record level in that each record can be read, processed, and written independently of all other records. The more PEs, the more records that can be simultaneously processed. Such an application is linearly scalable if the I/O is not a bottleneck.

Without global I/O, such an application must usually be implemented with a master/slave approach. The speed (or more typically, the number) of the master PE(s) must be adjusted to ensure that the slave PEs are not starved for data, and then readjusted whenever the number of slave PEs is changed. This style of programming is difficult to write, tune, and understand. In global I/O, because the "load balancing" is done automatically in the library, better performance and much greater code clarity are provided. Properties of global I/O:

1. All of the PEs must participate in establishing and terminating a unit/file connection for global I/O (i.e., the Fortran `OPEN` and `CLOSE` statements). Thereafter, PEs may independently participate, or not, in accessing the file.
2. Fortran `READ/WRITE` statements are synchronized and atomic. If multiple PEs are reading from the same file sequentially then each record will be read exactly once, although the order that the records are processed is nondeterministic. If multiple PEs write to the same file sequentially then all of the records will be written to the file, although again, there order is nondeterministic.

Input/output to/from shared variables is permitted on global I/O statements, as on private I/O statements, but no implicit synchronization (barrier) is performed to protect individual shared entities.

7 FUNCTIONS AND SUBROUTINES

Two opposing goals arise in designing the behavior of subroutines. High performance is crucial to the success of the CRAY T3D system but generality of operation, and specifically flexibility in passing parameters, is key to reducing the effort required by a programmer to use the machine effectively. This model adopts the principle that the system will generate the most efficient code possible from the available information. This allows the users to choose a course that best fits their needs when there is a conflict.

Subroutines may themselves in a sense be considered shared or private. A private subroutine is one that permits a task to function independently of all other tasks. It does not allocate shared data objects dynamically, including implicit redistribution. It may define or reference statically allocated shared objects, and it may accept shared dummy arguments. It may use locks, events, and critical sections. A shared subroutine is one that allocates shared data. This occurs when a routine allocates shared dynamic arrays, shared automatic arrays, or when an array is redistributed (see Section 7.1.1).

If private routines contain barriers, they are sometimes called *team routines*. Team routines must either ignore barriers (e.g., execute in a sequential region) or all tasks must execute the rou-

tine (to satisfy the barrier). This categorization of subroutines as shared or private is conceptual in nature and does not require the user to specify additional syntax to obtain one or the other. However, although calls to a shared subroutine are not required to arrive at the subroutine through the same call chain, the results are only defined if each task has the same sequence of shared subroutines in its call chain. Thus, one task may call subroutine A, which calls B, which calls a shared routine C, and another task may call A, which calls C directly. As long as B is not a shared subroutine the results are defined. But if one task calls a shared subroutine S, and another task calls a different shared subroutine T before it calls S, the results are undefined, and in fact the situation may cause deadlock or other unsavory behavior to occur because of problems with the use of shared resources. An ENTRY statement is not permitted in any routine that declares shared or UNKNOWN data.

7.1 Data Objects

Fortran 90 and Cray Fortran (cf77) both support several types of data objects within subroutines and functions. The list includes named and unnamed common blocks, dummy arguments, local scalars and arrays, and automatic arrays. For CRAFT Fortran, objects in each of these categories can be either shared or private with the exception of unnamed common blocks, which can only be private. The behavior of objects in common blocks is described in Section 2.5. Other object types are described in the following sections. An object that is referenced as private data in a routine must not be modified remotely until after the routine returns control to the caller.

Dummy Arguments

Dummy arguments may be declared as SHARED, PE_PRIVATE, UNKNOWN_SHARED, or UNKNOWN. When the declaration is shared or private, that declaration is honored and the code generated for references to that data is the most efficient possible with a redistribution upon entry and exit to the subroutine if necessary. The rules that apply to dummy and actual arguments are described in full later in this section, but a summary of the rules is:

1. If the dummy argument is declared SHARED then the dummy argument must not be an assumed size array.
2. If the dummy argument is private or it is shared and the actual argument matches the distribution exactly, no redistribution is done and the addressing scheme used is tailored for the declared distribution.
3. If the dummy argument is declared UNKNOWN_SHARED then a general addressing scheme is used and no redistribution is done.
4. If the dummy argument is declared UNKNOWN then access to the dummy argument is severely restricted and no redistribution is done.
5. If the dummy argument is declared private and the actual argument is shared, the addressing will assume that the dummy argument only references local data. In this case it is the user's responsibility to make sure the addressing schemes match. This allows the user to process local portions of shared arrays as if they were private. (See Section 2.6, "Shared to Private Coercion," for more information.)
6. If the dummy argument is declared SHARED and the actual argument is also SHARED but the two declarations do not match, the compiler will redistribute the actual argument to the distribution declared in the subroutine at its entry points and redistribute it to its original form at the subroutine exits. An assumed size array cannot be redistributed upon entry to a program unit. All tasks must participate in the redistribution. A routine that requires the redistribution of one or more objects cannot be called from within a master region.
7. If the dummy argument is declared SHARED or UNKNOWN_SHARED and the actual argument is declared private then the behavior is undefined.

It is worth noting that more information about array distributions causes subroutines to be more restrictive about how they are used, but it allows the compiler to take advantage of optimization opportunities that would otherwise not be available.

A dummy argument may have its distribution declared as being unknown. There are two varieties of unknown. The first, UNKNOWN, indicates that nothing is known about the argument, not even whether it is shared or private; it is not permitted to align a loop to an UNKNOWN argument. In addition, there are severe restrictions placed on

access to dummy arguments declared UNKNOWN. The arguments may only be accessed by special intrinsic routines that allow single element access. These intrinsics are:

```
CALL READ_UNKNOWN (V, A(I))
CALL WRITE_UNKNOWN (A(I), V)
```

where A is an array element or scalar and V is a value. The type of distribution may be discovered with the intrinsic

```
RESULT = IS_SHARED (A)
```

This intrinsic returns .TRUE. if shared, .FALSE. otherwise. The READ_UNKNOWN and WRITE_UNKNOWN routines and the IS_SHARED function must appear on an INTRINSIC statement before use. This intrinsic is useful because a single entry point may query the distribution of a dummy argument and call an appropriate routine based on the result. An UNKNOWN dummy argument may be passed to another subprogram, but only if the entire object is passed. The syntax for this declaration is

```
CDIR$ UNKNOWN arg1 arg2, . . . , argn
```

The second, UNKNOWN_SHARED implies that the distribution is not known and no redistribution should be done, but that it is dimensionally distributed data. There are no restrictions on the use of data declared UNKNOWN_SHARED. The syntax for this declaration is

```
CDIR$ UNKNOWN_SHARED arg1, . . . , argn
```

When the distribution is unknown at compile time it is determined at run-time. The subroutine assumes the most general dimensional distribution possible, which often causes access to the dummy argument to have lower performance.

Local Objects

Subroutines may declare local variables, i.e., objects local to a subroutine that are shared or private. Both arrays and scalars may be declared as shared or private. Data initialization rules for local objects follow the rules for cf77, i.e., statically allocated arrays and scalars may be initialized with DATA statements. Shared and private local objects follow the rules for shared and private objects outlined in Section 2. Subroutines that allocate shared local objects (explicitly or implicitly) contain implicit barriers on entry to and exit from each subroutine and are called shared subroutines.

Automatic Arrays

Both shared and private automatic arrays are supported in this model. Subroutines that allocate shared automatic arrays implicitly contain barriers on entry to and exit from the routine and are called shared subroutines. This is required to maintain consistent memory allocations across all processors. When a subroutine is called that allocates a shared automatic array, all processors must request the same sized allocation for each shared automatic array. Private automatic arrays may vary in size from task to task without difficulty.

7.2 Pointers

The terms *shared pointer* and *private pointer* are ambiguous. The ambiguity arises because there are two data objects, the pointer itself and the object being pointed to. The term shared pointer means that the pointer is pointing to shared data, and private pointer means the pointer is pointing to private data. The pointer itself cannot be shared. That is, the following is an error:

```
      POINTER ( PT, B )
      REAL B(1024)
CDIR$ SHARED PT      ! Error -  the pointer
                     ! must be pe_private
```

The points themselves are always private. The pointee array (B) can be either shared or private.

Declaring Fortran Pointers

The declaration of a private pointer is no different than the current implementation in cf77. The following declaration:

```
POINTER (P1, A1)
REAL A1(1000, 1000)
```

declares a private pointer P1 whose pointee array is A1. The current heap allocation routines (e.g., HPALLOC and HPDEALLC) provide access to the private heap.

A shared pointer is declared by adding the SHARED directive to the pointee array. For instance:

```
      POINTER (P2, A2)
      REAL A2(1024, 1024)
CDIR$ SHARED A2(:BLOCK, :BLOCK)
```

declares a shared pointer P2 whose pointee array A2 has a known distribution.

Shared Pointers

The usage of shared pointers is more restricted than private pointers. No pointer arithmetic is allowed, only one pointee can be specified for each shared pointer (i.e., multiple pointees are not allowed), and the only allowable operations are:

```
CALL SHLOC(ptr, ary)
CALL SHMALLOC(pointer, istatus)
CALL SHMALLOC(pointer, istatus, length)
CALL SHFREE(pointer)
```

The shared pointer routines must appear on an INTRINSIC statement before use.

SHLOC assigns the shared address of ary to ptr. The pointer argument must be a shared pointer and the array must also be shared. If the first argument to SHLOC has a pointee array with a known distribution (P2), then the distribution of the second argument must match. For example:

```
      POINTER (Q,X)
      REAL X(128,128), Y(128,128)
      REAL Z(128,128)
CDIR$ SHARED X(:BLOCK, :BLOCK)
CDIR$ SHARED Y(:BLOCK, :BLOCK)
CDIR$ SHARED Z(:BLOCK, :)
C     OK
      CALL SHLOC(Q, Y)
C     Error - not array base
      CALL SHLOC(Q, Y(2))
C     Error - distribution mismatch
      CALL SHLOC(Q, Z)
```

SHMALLOC and SHFREE provide access to the shared heap. Only pointers whose pointee array has a known distribution (Q) may be used in calls to SHMALLOC or SHFREE. SHMALLOC can get the size of the space it needs to allocate from information contained in the pointer. In this case it is sufficient to pass a single argument (the pointer) to SHMALLOC. If the allocated size is different than the size of pointee array, two arguments are passed to SHMALLOC. The following example shows a possible use of having a different size:

```
      POINTER (R, W)
      REAL W(128, 10000000)
CDIR$ SHARED W(:BLOCK, :)
      CALL SHMALLOC(R, ISTAT, 128*100)
      ... W(I,100) ...
```

Because the last dimension of pointee array W is degenerately distributed, it need not be a power of 2. Because there is no actual storage allocated for pointee arrays, their size can be the largest that will ever be allocated. However, if the pointer R were passed as the only argument, too much space would be allocated because SHMALLOC would extract the total array size (128*10000000) from information in pointer R. SHFREE returns shared heap space to the shared heap.

It is important to understand that a redistributed dummy argument cannot be referenced by a pointer that points to the original actual argument. The pointer has the address and distribution of the array prior to redistribution. For example:

```
      SUBROUTINE STEVE(X,N)
      REAL X(N)
      COMMON /XXX/ P
      POINTER (P, PA)
      REAL PA(1024)
CDIR$ SHARED PA(:BLOCK(2))
CDIR$ SHARED X(:BLOCK(4))
C     implicit redistribution occurs
      ...
      PA(I) = 2.0
C     Error, referencing redistributed actual
      ...
      END
      SUBROUTINE BUD()
      COMMON /XXX/ P
      POINTER (P, PA)
      REAL PA(1024)
CDIR$ SHARED PA(:BLOCK(2))
      CALL SHMALLOC(P, ISTAT)
C     allocate shared heap space
      CALL STEVE(PA, 1024)
      END
```

This kind of aliasing through pointer P results in undefined behavior.

7.3 Assumed Size Arrays

The following restrictions are placed on assumed size arrays.

1. Pointee arrays associated with shared pointers cannot be declared as assumed size arrays.
2. If a dummy argument is declared to be an assumed size array then it must not be explicitly shared with a SHARED directive (it can be implicitly shared with an UNKNOWN_SHARED directive).
3. Assumed size arrays cannot be used in BLKCT, LOWIDX, HIIDX, and PES functions (defined in Section 8.2) if the second argument represents the assumed size dimension (i.e., last dimension).
4. An assumed size array cannot appear in a COPY clause of an END MASTER directive.

8 INTRINSIC FUNCTIONS

This programming model offers a variety of intrinsic functions intended to support both high level and low level programming. The high level functions support basic data parallel operations. Data parallel operations are used when a programmer defines a program from the perspective of what the system as a whole will do.

Low level intrinsic functions give detailed information about how shared data are distributed across the available processors. Other functions give information about the relationship between tasks based on the distribution of data. Two additional functions provide broadcast and multicast capabilities. All intrinsic functions must appear in an INTRINSIC statement (as is required by Fortran.)

8.1 Data Parallel Functions

The data parallel functions supported consist of three basic function families, namely reduction functions, parallel prefix functions, and segmented scan functions. Thomas Leighton [10] provides a thorough treatment of their uses in a massively parallel processor (MPP) environment. Reduction functions are the most widely recognized of the three. They include array summation, product, maximum or minimum value, and the location (array index) of the maximum or minimum value. Parallel prefix functions are a generalization of reductions that retain an ordered list of partial reductions. Segmented scan functions are a generalization of parallel prefix functions; they offer the ability to perform many prefix operations over subranges of an array. This makes them useful for solving linear algebra problems with some representations of sparse matrices. Some representations where scan functions are useful are banded, sky line, and block tridiagonal sparse matrices.

Reduction Functions

Six reduction functions are supported: SUM, PRODUCT, MINVAL, MAXVAL, MINLOC, and MAXLOC. SUM adds the elements in the specified array. PRODUCT multiplies the elements. MINVAL finds the minimum value. MAXVAL finds the maximum value in an array. MINLOC finds the location (array index) of the minimum value. MAXLOC finds the location of the maximum value.

Each of these functions has the same syntax, which is modeled after the syntax used in Fortran 90. This document will describe the syntax for one reduction function, SUM, and note only that it is the same for the other five. (MINLOC and MAXLOC do not have a DIM argument.)

```
RESULT = SUM(ARRAY, DIM, MASK)
```

DIM and MASK are optional arguments. SUM adds all the elements along the dimension DIM corresponding to the .TRUE. elements of MASK. When MASK is not specified, all elements are summed. When DIM is not specified, all elements are reduced to a single scalar value. When DIM is specified, the result is an array whose rank is 1 smaller than the rank of ARRAY.

Parallel Prefix Functions

Parallel prefix functions behave similarly to reduction functions, but they retain all partial reductions. Let $\oplus$ be some associative binary operation (e.g., addition, multiplication, minimum value, maximum value, minimum value location, or maximum value location). Mathematically, parallel prefix may be expressed as

```
RESULT(1:N) = prefix (⊕, A(1:N))
```

which means that

```
RESULT(1)  = A(1)
RESULT(2)  = A(1)⊕A(2)
RESULT(3)  = A(1)⊕A(2)⊕A(3)
RESULT(4)  = A(1)⊕A(2)⊕A(3)⊕A(4)
...
RESULT(N)  = A(1)⊕A(2)⊕A(3)⊕A(4)⊕...⊕A(N)
```

Notice that unlike reductions, this meta-function (or higher-order function) always requires that RESULT and A are conformable—RESULT will never be a scalar.

This model supports a family of parallel prefix functions that is similar to the family of reduction functions. Supported functions include the prefix equivalents of SUM, PRODUCT, MAXVAL, and MINVAL. There are no plans to support an equivalent to MAXLOC or MINLOC. The functions are called PRESUM, PREPROD, PREMAX, and PREMIN. As before, only the prefix sum function is defined in detail and other prefix functions are defined by extrapolation.

```
RESULT = PRESUM(ARRAY, DIM, MASK)
```

DIM and MASK are optional arguments. The result is always an array of the same shape and size as ARRAY. PRESUM computes partial sums for all the elements along dimension DIM corresponding to the .TRUE. elements of MASK. When MASK is not specified, all elements are used in the partial sum. When DIM is not specified, all elements are used, innermost dimensions first. When DIM is specified, the prefix operation occurs only along the specified dimension.

Segmented Scan Functions

Scan functions behave similarly to parallel prefix functions, but they carry an additional mask of stop bits that define where each prefix begins and ends. Supported is a family of scan functions, called SCANSUM, SCANPROD, SCANMAX, and SCANMIN. This may be illustrated by

```
RESULT(1:N) = scan(⊕, A(1:N), STOP(1:N))
```

Assume that STOP(1:N) contains

```
0, 0, 0, 0, 1, 0, 0, 0, ..., 0
```

The scan means that

```
RESULT(1)  = A(1)
RESULT(2)  = A(1)⊕A(2)
RESULT(3)  = A(1)⊕A(2)⊕A(3)
RESULT(4)  = A(1)⊕A(2)⊕A(3)⊕A(4)
RESULT(5)  = A(1)⊕A(2)⊕A(3)⊕A(4)⊕A(5)
RESULT(6)  = A(6)
RESULT(7)  = A(6)⊕A(7)
RESULT(8)  = A(6)⊕A(7)⊕A(8)
...
RESULT(N)  = A(I+1)⊕A(I+2)⊕A(I+3)⊕...⊕A(N)
```

Notice that unlike reductions, but similarly to prefix operations, this meta-function (or higher-order function) always requires that RESULT and A are conformable—RESULT will never be a scalar.

This model supports a family of segmented scan functions that is similar to the family of parallel prefix functions. Supported functions include the scan equivalents of PRESUM, PREPROD, PREMAX, and PREMIN. There are no plans to support an equivalent to MAXLOC or MINLOC. The functions are called SCANSUM, SCANPROD, SCANMAX, and SCANMIN. As before, only the scan sum function is defined in detail and other scan functions are defined by extrapolation.

```
RESULT = SCANSUM(ARRAY, STOP, DIM, MASK)
```

DIM and MASK are optional arguments. The result is always an array of the same shape and size as ARRAY. SCANSUM computes partial sums for all the elements along dimension DIM corresponding to the .TRUE. elements of MASK. When MASK is not specified, all elements are used in the partial sum. When DIM is not specified, all elements are used, fastest running dimensions first. When DIM is specified, the scan operation occurs only along the specified dimension.

Semantics of Data Parallel Functions

Data parallel functions have semantics as follows:

1. When executing in a master region or a shared loop, one task does all the work.
2. When executing in a parallel region, the behavior varies slightly depending on the attributes of RESULT and ARRAY.
3. When RESULT and ARRAY are both shared, ARRAY is processed by work-sharing and the value is stored in RESULT.
4. When RESULT is private and ARRAY is shared, ARRAY is processed by work-sharing and the value is broadcast to all copies of RESULT that participated in the function.
5. When RESULT is a shared scalar object and ARRAY is private, a race condition exists and the value of RESULT is undefined unless

ARRAY has the same value on all PEs. For the reduction functions, when only one task participates, or each task references a different shared value, as in:

```
RESULT(MY_PE()) = SUM(ARRAY)
```

the result is defined and each referenced RESULT(I) contains the result of the local summation.

6. When RESULT and ARRAY are both private, the function takes on its sequential semantics and the function is performed locally by each task participating in the function.

Note that when RESULT is shared and it executes in a parallel region, all tasks must participate and a barrier synchronization is implied.

8.2 Data Mapping Functions

The data distribution mechanism described in Section 2 provides an important level of abstraction. Because the implementation of this mechanism is sometimes subtle, this model supports several data mapping functions to make low level programming more accessible to the user. The data mapping functions provide the user with low level access to the data distribution mechanism used by this model.

Five functions are supported. Three of them provide direct access to the blocks of data distributed by the mechanism. Two of the functions provide information about how data are mapped to the processors. A detailed description of each function follows.

BLKCT(A, D, P) returns an integer that represents the number of blocks of elements in the Dth dimension of array A that are resident on processor P. A must not be an assumed size array if D represents the last dimension.

```
CT = BLKCT(A, D, P)
```

LOWIDX(A, D, P, K) returns an integer that represents the lowest index of block K in the Dth dimension of array A that is resident on processor P. A must not be an assumed size array if D represents the last dimension.

```
LOW = LOWIDX(A, D, P, K)
```

HIIDX(A, D, P, K) returns an integer that represents the highest index of block K in the Dth dimension of array A that is resident on processor P. A must not be an assumed size array if D represents the last dimension.

```
HI = HIIDX(A, D, P, K)
```

As an example of how this might be used, the following code references exactly the elements of an array that are resident on the PE that executes the code.

```
      DIMENSION X(1024)
CDIR$ SHARED X(:BLOCK(32))
CDIR$ PE_RESIDENT X
      INTRINSIC BLKCT, LOWIDX, HIIDX
      ...
      DO K = 1, BLKCT(X, 1, MY_PE())
         LO = LOWIDX(X, 1, MY_PE(), K)
         HI = HIIDX (X, 1, MY_PE(), K)
         DO I = LO, HI
            ... X(I) ...
         END DO
      END DO
```

HOME(X) returns an integer that represents the processor on which X resides. X is a scalar object or array element rather than an entire array.

```
HOME(X)
```

PES(A, D) returns an integer that represents the number of processors used in the Dth dimension of array A. A must not be an assumed size array if D represents the last dimension.

```
PRCT = PES(A, D)
```

The data mapping functions must appear on an INTRINSIC statement before use.

9 EXAMPLE

It is difficult to gauge the ease of use of a programming model without some specific examples of its use in actual codes. This programming model was written with the goal of easily modifying existing Fortran codes to get a very good speedup. A companion goal was to make it possible for the user to gain incremental improvement with small additional effort.

Because real programs cannot be shown here, a smaller example is shown to demonstrate the directives in the model. A standard matrix multiply subroutine is first decorated with directives in a

straightforward manner. The resultant code will run well in parallel. These changes are entirely mechanical and require very little analysis. A further set of changes that requires some knowledge of the code is added. These will further improve the code by increasing data locality.

```
SUBROUTINE MATMUL(A, B, C, L, M, N)
DIMENSION A(L,M), B(M,N), C(L,N)
DO K=1,N
 DO I=1,L
   DO J=1,M
     C(I,K) = C(I,K) + A(I,J) * B(J,K)
    ENDDO
  ENDDO
ENDDO
END
```

The most useful thing to do to achieve high performance is to spread the data and work across the machine. This helps both with data locality and gives a large address space for big problems. The SHARED directive spreads the data around the machine, the DOSHARED directive spreads the work. As mentioned earlier, data locality is key to high performance. Therefore, the work will be aligned to the location of the data. Finally to increase data locality, The (:BLOCK, :BLOCK) distribution has been chosen here for C, (:BLOCK, :) for A because whole rows of A are used at a time, and (:, :BLOCK) for B for analogous reasons.

```
      SUBROUTINE MATMUL(A, B, C, L, M, N)
      DIMENSION A(L,M), B(M,N), C(L,N)
CDIR$ SHARED A(:BLOCK, :),B(:, :BLOCK)
CDIR$ SHARED C(:BLOCK, :BLOCK)
CDIR$ DOSHARED (K,I) ON C(I,K)
      DO K=1,N
         DO I=1,L
            DO J=1,M
               C(I,K) = C(I,K) + A(I,J)
     *          * B(J,K)
            ENDDO
         ENDDO
      ENDDO
      END
```

This example, with minimal changes, now has data spread across the processors and work shared among the processors. These data are computed in as local a context as possible.

10 CONCLUSIONS

In this report we have described a highly flexible programming model. This model unifies several divergent programming styles, including message-passing, work-sharing, and data parallel. Data may be stored in distributed shared memory or memory that is private to a task. The work may be controlled on a task-by-task basis, or work can be shared among multiple tasks through work-sharing constructs. Input and output can be performed by a single task or by many tasks. This model also includes a rich set of synchronization primitives. When desired, all of these features can be used together in the same program.

It has been clear throughout the design of this programming model that users have an enormous variety of often conflicting needs. For some, the ability to exploit the available data locality and data cacheability is of utmost importance. For others the ability to use all available memory and access it rapidly provides the greatest benefit. And although it is often overlooked, I/O is a major bottleneck for a number of important commercial codes. This list of design problems is necessarily short and very incomplete. Indeed, every new major code seems to bring a new set of problems that, if addressed, would provide some important improvement *for that code*. Yet for this model to be accepted by the scientific computing community, it had to be similar to what they were already using or they would not have the time to learn how to use it effectively! Balancing the desires of fast, flexible, safe, small, and familiar is difficult, especially when it must be done within a commercially feasible time frame with limited resources. To achieve this trade off, our emphasis has been on speed and simplicity first. We have incorporated additional features only when doing so has provided a real, identifiable benefit without degrading the quality of existing features.

A compiler for this programming model is currently under development and scheduled for release in 1994. As such, reliable performance numbers are not available at the time of this writing, but preliminary performance numbers are highly encouraging. The CRAFT compiler is built using Cray's mature `cf77` compiler as a base, and as such will have the benefit of a reliable, highly optimizing Fortran compiler to provide syntactic, semantic, and dependence analysis. CRAFT requires some additional analysis above and beyond what `cf77` already provides, and a code generator

to support the new instruction set. But much of what is needed, including many optimizations, is common to the parallel–vector compiler.

The core definition of CRAFT is now complete, although a lot of work remains to be done. Some of the features proposed for future releases are: relaxing the restrictions on power-of-2 array sizes and distributions, canonical distributions, additional loop distribution mechanisms, and task teams. Allowing the user to distribute arrays with arbitrary sizes will permit significantly more efficient and flexible memory utilization. The cost will be less efficient memory references when such arrays are used. Canonical distributions strike a balance between the storage and sequence association properties of traditional Fortran data objects and the control over data locality provided by (dimensional) distributions. Canonical distributions scatter array cells in a fixed manner (e.g., cycle blocks of four words across PEs) after array index linearization has taken place. New loop distribution mechanisms may include adaptive mechanisms, perhaps some adaptation of guided self-scheduling [11] appropriate to MPPs. Last of all, task teams would support explicit functional decomposition, which would assist the user in dividing up tasks to handle functionally separate computations while allowing each task team to pursue work-sharing computations among tasks within the team.

ACKNOWLEDGMENTS

The work represented by this article was supported by The Defense Advanced Research Projects Agency under Agreement No. MDA 972-92-H-0002 dated January 21, 1992.

REFERENCES

[1] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, V. Sunderam, "A User's Guide to PVM-Parallel Virtual Machine," Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

[2] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, M.-Y. Wu, *Fortran D Language Specification.* Houston, TX: Rice University, 1991.

[3] B. Chapman, P. Mehrotra, H. Zima, "Programming in Vienna Fortran," *Scientific Programming*, Vol. 1, pp. 31–50, 1992.

[4] Cray Research, *CF77 Compiling System, Volume 4: Parallel Processing Guide, SR-3071 4.0.* Eagan, MN: Cray Research, Inc., 1991.

[5] Cray Research, *CRAY Y-MP, CRAY X-MP EA, and CRAY X-MP Multitasking Programmer's Manual,* SR-0222 F-01. Eagan, MN: Cray Research, Inc., 1991.

[6] Thinking Machines Corporation, *CM Fortran Reference Manual.* Cambridge, MA: Thinking Machines Corporation, 1991.

[7] BBN Advanced Computers Inc., *TC2000 Fortran Language Reference.* Boston, MA: BBN Advanced Computers Inc., 1990.

[8] K. Warren, B. Gorda, E. D. Brooks III, "Programming in PFP," Technical Report UCRL-MA-107028, Lawrence Livermore National Laboratory, April 1991.

[9] "High Performance Fortran Language Specification", *High Performance Fortran Forum, Scientific Programming*, Vol. 2, Nos. 1 & 2, 1993.

[10] F. Thomson Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* San Mateo, CA: Morgan Kaufman Publishers 1992.

[11] C. Polychronopolous and D. Kuck. "Guided self-scheduling: A practical scheduling scheme for parallel computers," IEEE Trans. Comput., vol. 36, December 1987.

[12] International Standards Organization, *ISO/IEC 1539:1991, Information Technology-Programming Languages–Fortran.* Geneva: ISO, 1991.

APPENDIX 1 PROGRAMMING MODEL DIRECTIVES AND FUNCTIONS

This section summarizes the syntax of CRAFT directives and intrinsic functions.

Note: The notation var_i represents the name of a data object, which may be the name of a scalar variable or an array. The notation α_i represents a keyword that can be `:BLOCK` or `:BLOCK(N)`, optionally with weights, or a colon by itself `(:)`. The expression $(\alpha_1, \alpha_2, \ldots, \alpha_r)$ may only be used with array names or geometry labels. The notation `[::]` indicates that "`::`" is optional. The notation $f_i(\bar{I})$ indicates an index expression of the form $a_i I_i + b_i$ where a_i and b_i are integers.

1. Data objects

```
CDIR$ PE_PRIVATE var1, var2, ..., varn
CDIR$ SHARED var1 (α1, α2, ..., αr), ...
CDIR$ GEOMETRY geom1 (α1, α2, ..., αr), ...
CDIR$ SHARED (geom) [::] var1, var2, ..., varn
CDIR$ UNKNOWN var1, var2, ..., varn
CDIR$ UNKNOWN_SHARED var1, var2, ..., varn
```

2. Sequential execution

```
CDIR$ MASTER
CDIR$ END MASTER [, COPY(var1, var2, ..., varn)]
```

3. Loops

```
CDIR$ DOSHARED (I1, I2, ..., In) ON A (f1 (Ī), f2 (Ī), ..., fr (Ī))
CDIR$ NO BARRIER
```

4. Synchronization primitives

```
CDIR$ BARRIER
      CALL BARRIER()
      CALL SET_BARRIER()
      CALL WAIT_BARRIER()
      IB = TEST_BARRIER()
      CALL SET_LOCK(lock)
      CALL CLEAR_LOCK(lock)
      L = TEST_LOCK(lock)
      CALL SET_EVENT([event])
      CALL WAIT_EVENT([event])
      CALL CLEAR_EVENT([event])
      S = TEST_EVENT([event])
CDIR$ CRITICAL
CDIR$ END CRITICAL
CDIR$ ATOMIC UPDATE
```

5. Subroutine arguments

```
CDIR$ UNKNOWN var1, var2, ..., varn
CDIR$ UNKNOWN_SHARED var1, var2, ..., varn
CDIR$ PE_RESIDENT var1, var2, ..., varn
```

6. Intrinsic functions. The following functions must appear on an INTRINSIC statement before use.
   A. Reduction functions

```
RESULT = SUM(ARRAY, DIM, MASK)
RESULT = PRODUCT(ARRAY, DIM, MASK)
RESULT = MINVAL(ARRAY, DIM, MASK)
RESULT = MAXVAL(ARRAY, DIM, MASK)
RESULT = MINLOC(ARRAY, MASK)
RESULT = MAXLOC(ARRAY, MASK)
```

B. Parallel prefix functions

```
RESULT = PRESUM(ARRAY, DIM, MASK)
RESULT = PREPROD(ARRAY, DIM, MASK)
RESULT = PREMIN(ARRAY, DIM, MASK)
RESULT = PREMAX(ARRAY, DIM, MASK)
```

C. Segmented scan functions

```
RESULT = SCANSUM(ARRAY, STOP, DIM, MASK)
RESULT = SCANPROD(ARRAY, STOP, DIM, MASK)
RESULT = SCANMIN(ARRAY, STOP, DIM, MASK)
RESULT = SCANMAX(ARRAY, STOP, DIM, MASK)
```

D. Data mapping functions

```
CT   = BLKCT(A, D, P)
LOW  = LOWIDX(A, D, P, K)
HIGH = HIIDX(A, D, P, K)
HOME = HOME(X)
PES  = PES(A, D)
```

E. Query functions

```
RESULT = IN_PARALLEL()
RESULT = IN_DOSHARED()
RESULT = IS_SHARED(A)
```

F. Data access routines

```
CALL READ_UNKNOWN(V, A(I))
CALL WRITE_UNKNOWN(A(I), V)
```

G. Task identity function

```
RESULT = MY_PE()
```

H. Shared pointer routines

```
CALL SHLOC(ptr, istatus, ary)
CALL SHMALLOC(ptr, istatus)
CALL SHMALLOC(ptr, length)
CALL SHFREE(ptr)
```

APPENDIX 2 FORTRAN 90 FEATURES

Fortran 90, as an emerging standard that has applicability to both scientific computing and massively parallel computing, has many features that are of interest. Although it was not an option to implement a full Fortran 90 compiler for the CRAY T3D system in the time allowed, two basic elements of the Fortran 90 language will be available to CRAFT users. The first is array assignment statements, e.g., `A(1:N) = F(B(1:N))`. The second is the WHERE statement.

It is anticipated that many of the most important Fortran 90 intrinsic functions will also be implemented for CRAFT. The Fortran 90 intrinsic functions that will receive earliest consideration for implementation are shown in Table 1.

Complete definitions of these intrinsic functions, array assignment, and WHERE statements can be found in the Fortran 90 language standard [12].

Table 1. Fortran 90 Functions

ALL	DOT_PRODUCT	MAXVAL	PACK	SUM
ANY	EOSHIFT	MERGE	PRODUCT	TRANSPOSE
COUNT	MATMUL	MINLOC	RESHAPE	UNPACK
CSHIFT	MAXLOC	MINVAL	SPREAD	