The Cray Research CRAFT-90 Programming Model

Tom MacDonald, Compiler Group, Cray Research, Inc., Eagan, MN

ABSTRACT: *In their current Fortran product CF77™, Cray Research, Inc. has defined a Fortran based programming model for the Cray T3D™ system that allows programmers to specify both data sharing and work sharing. Analysis of this programming model identifies key language features to be retained and others that should be eliminated. Because future Fortran programming directions are based on CF90™, this analysis helped determine how to refine and improve the model for CF90™.*

1 Introduction

The Cray Research, Inc. (CRI) MPP Fortran Programming Model, CRAFT-77, defines a feature rich implementation that caters to a diverse programming community still exploring how to best exploit a relatively new technology called massively parallel processing. Analysis of this initial implementation allowed CRI to draw some conclusions about the effectiveness of the CRAFT-77 extensions. These conclusions are used in an ongoing decision making process to determine what features to retain, change and eliminate in a follow-on implementation called CRAFT-90. The final CRAFT-90 features will be implemented in the CF90 product for the next generation MPP, the Cray T3E™ system. There is still further refinement needed before a final language finished.

2 Architecture Assumptions

Some assumptions about the underlying hardware architecture are present in CRAFT-77 and remain in CRAFT-90. The first assumption is that the architecture is a Multiple Instruction Multiple Data (MIMD) parallel architecture. These architectures use a network to combine several general purpose Processing Elements (PE) that are capable of executing their own separate programs containing their own data. MIMD architectures require synchronization primitives to control communication and coordination among the PEs.

These programming models assume that each PE has its own privately accessible memory module and other accessible memory modules that are *remote.* Remote references traverse an interconnection network to some uniquely identifiable memory on the network. This is called a *distributed memory* system. A high speed interconnection network is critical to providing high performance.

Although the memory system is physically distributed across the PEs, there is architectural support for viewing all memory systems as a *global address space* with nonuniform access time. The architecture also provides a *memory latency tolerance* feature that decreases the marginal cost of referencing a block of remote memory locations. This allows prefetching of data for use later in a computational operation, making it easier to hide the latency associated with remote references.

Finally, the architecture provides the necessary primitives for high level synchronization support.

3 Explicit Communication Libraries

There are really two different styles of programming available with the CRI MPP Fortran Programming Model. The first style is *explicit communication* defined by a set of library functions used for communication and synchronization. The PVM message passing and the shared memory library *(SHMEM)* are two explicit communication libraries supported on the Cray T3D.

Good progress in portable, high performance, parallel programming has been made with the *de facto* message-passing standard, PVM. MPI message-passing is a likely successor. CRI's SHMEM library provides asynchronous, low latency, and high bandwidth *get* and *put* operations that directly support the globally addressable memory of the Cray T3D. This means the SHMEM library routines typically offer higher performance than PVM.

A pure message passing program does not require a global address space because communication is synchronized through a library. All communication must be explicitly coordinated within the application with one PE sending data and another PE receiving. The SHMEM library routines are able to provide explicit communication through globally accessible variables. A single PE is able to both store (**-SHMEM_PUT**) and load **(SHMEM_GET)** data associated with another PE without any

assistance from the other PE. Synchronization is explicitly placed in the program before the transmitted data is referenced.

Since programming with explicit communication can be tedious, CRAFT-77 was defined to permit implicit communication through shared variables. Both implicit and explicit communication are permitted in the same program.

4 CRAFT-90

CRAFT-90 maintains the same look and feel as CRAFT-77 in that a global address model, data distribution, and work sharing are major components of both implementations. Both implementations employ a Single Program Multiple Data (SPMD) execution model that permits explicit communication styles and implicit global address styles to be combined in a single program. The execution model, therefore, is the same for both styles of communication, and encourages the user to experiment with using both of them.

CRAFT-90 retains the notion of shared arrays that are distributed across the processors, and allows each dimension to be distributed independent of the others. Some of the data distribution features available in CRAFT-77 have been removed from CRAFT-90. An important enhancement to CRAFT-90 is the elimination of the requirement that distributed array extents be a power-of-two. The power-of-two restrictions in CRAFT-77 are too restrictive and prevent many users from effectively using CRAFT-77.

The work sharing constructs have been retained including the **DOSHARED** directive and array syntax involving shared arrays. The Fortran 90 array intrinsics (e.g., **CSHIFT)** still accept shared array arguments.

5 N$PES and MY_PE

There are two special identifiers supported in CRAFT-77, **N$PES** and **MY_PE.** Both of these identifiers are retained in CRAFT-90. The identifier **N$PES** is a special constant whose value is the total number of PEs assigned to the execution of the program (which is also the total number of tasks). It may be used most places an integer constant is required. The **N$PES** feature allows data structures, and thus the size of the problem being solved, to scale with the number of PEs assigned to the program.

The Fortran standard requires constants in some situations. The following are some Fortran statements that require constant expressions along with a comment field indicating if special **N$PES** constant is allowed in that statement.

```
PARAMETER (P=N$PES, Q=N$PES/2+4)   !OK
DIMENSION A(100, N&PES)            !OK
CHARACTER *(2*N$PES), CHVAR        !Error
INTEGER IX(N$PES), IY(N$PES)       !OK
DATA IX /N$PES*N$PES/              !Error
Z = (N$PES,N$PES)                  !Error
```

This shows that the **N$PES** constant is not allowed on **CHARACTER or DATA** statements; nor can it be used in a complex constant. It is supported on **PARAMETER** and **DIMENSION** statements.

In order to support a general model, the value of **N$PES** does not need to be fixed at compile time. This flexibility is the reason the above restrictions are imposed on constant expressions involving the **N$PES** constant. The value of **N$PES** can be fixed at compile, link, or execution time. If fixed at execution time, the program is automatically re-linked just prior to execution.

Each *task* executing within a program has a unique identification. (There is a one-to-one correspondence between tasks and PEs.) The name given to each task is referenced by using the intrinsic function **MY_PE()**. It evaluates to an integer value between **0** and **N$PES-l** inclusive. It can be used anywhere an intrinsic function is permitted.

6 Data Objects

Data obiects are assigned to storage locations on the distributed memory modules associated with each PE. They include variables, common blocks, and dynamically allocated stack and heap space. The global address model supports two additional attributes for data objects. Data objects are either *private* or *shared.* CRAFT-90 retains these two additional attributes. Data distributions allow programmers to distribute their data objects in a variety of ways. These data distribution features can help increase performance by decreasing the number of remote references, and by increasing *locality,* that is, increasing the number of references local to a PE.

Private data is replicated on all tasks. Each executing task owns its own copy of the data object. The default attribute is the private attribute. All data objects are assumed to be private unless explicitly declared otherwise. Variables may also be explicitly declared private with the following directive:

CDIR$ PE_PRIVATE var_1, var_2, ..., var_n

The directive is named **PE_PRIVATE** to avoid confusion with Fortran 90's notion of private to a module. In this paper, the term *private is* used to mean private to a task or PE. In CRAFT-77, a private data object can only be referenced by the task that owns it. A reference to a private data object is, therefore, never a remote reference.

This restriction is being removed in CRAFT-90. A private data object can be referenced remotely if the base address of the data object is consistent across all the PEs. Additional features or semantics need to be defined to help users guarantee consistent addressing for specified private data objects.

6.1 Shared Data Objects

A shared data object can always be referenced by any task. Shared data is not replicated, in that there is only one data object shared by all tasks. Shared data objects must be declared as such, and can be distributed across all of the PEs executing the program. Although entire common blocks cannot be distributed, the individual entities within a common can. Similarly,

local variables and dummy arguments can also be distributed. The one exception is entities declared within *blank common blocks,* because the size of these data objects can expand. CRAFT-77 allows different dimensions of a shared array to be distributed differently. The declaration of a *dimensionally distributed* array specifies the distribution for each dimension. Each dimension is distributed as if it were independent from all other dimensions. Dimensional distributions are intended to increase the locality of data references by providing control over the way elements of an array are distributed.

CRAFT-90 retains the dimensional distribution capability but does not provide all of the generality of CRAFT-77. Little used features and those features that are not performance oriented have been eliminated. The major feature being added is the removal of CRAFT-77's power-of-two limitations for array extents. The power-of-two requirements in CRAFT-77 are too restrictive, and too big an impediment to using CRAFT for many applications.

The following list shows the dimensional distributions supported in CRAFT-77.

:BLOCK(N) a *block* distribution where *N* contiguous elements are in each block and each PE owns the same number of blocks,

:BLOCK a *block* distribution where each PE owns exactly one block of contiguous elements, and

: the *degenerate* distribution where each PE owns an entire dimension (i.e., not distributed).

From the above list, the **:BLOCK(N)** distribution has been removed from CRAFT-90 because it can introduce lower performance. The lower performance is a result of the complicated addressing involved with referencing a particular remote element from an array distributed **:BLOCK(N),** and the increased complexity of locality analysis. This lower performance resulted in few users willing to use this distribution method.

The following example shows a single dimensioned array that is distributed with a :**BLOCK** distribution across 4 PEs.

```
REAL X(20)
CDIR$ SHARED X(:BLOCK)
```

The **:BLOCK** distribution is retained in CRAFT-90.

The degenerate distribution allows an entire dimension to be assigned to one PE. This is useful when, for example, it is beneficial to assign an entire row or column of an array to one PE. For example:

```
REAL Y(3,20)
CDIR$ SHARED Y(:,:BLOCK)
```

declares the first dimension to always reside on a single PE. The following figure shows this distribution across four PEs.

The above example declares a “block” assigned to each PE. It is also possible to have just one column assigned to a PE with the following declaration:

Distribution Across 4 PEs									
PE0					PE1				
1	2	3	4	5	6	7	8	9	10

Distribution Across 4 PEs									
PE2					PE3				
11	12	13	14	15	16	17	18	19	20

Distribution Across 4 PEs									
PE0					PE1				
1, 1	1, 2	1, 3	1, 4	1, 5	1, 6	1, 7	1, 8	1, 9	1,10
2, 1	2, 2	2, 3	2, 4	2, 5	2, 6	2, 7	2, 8	2, 9	2,10
3, 1	3, 2	3, 3	3, 4	3, 5	3, 6	3, 7	3, 8	3, 9	3,10

Distribution Across 4 PEs									
PE2					PE3				
1,11	1,12	1,13	1,14	1,15	1,16	1,17	1,18	1,19	1,20
2,11	2,12	2,13	2,14	2,15	2,16	2,17	2,18	2,19	2,20
3,11	3,12	3,13	3,14	3,15	3,16	3,17	3,18	3,19	3,20

```
REAL Z(100,N$PES)
CDIR$ SHARED Z(:,:BLOCK)
```

Since this declares the first dimension to be degenerate, and the second dimension's size to be the number of PEs, each PE is assigned a column of array **z**.

The degenerate distribution method is also retained in CRAFT-90.

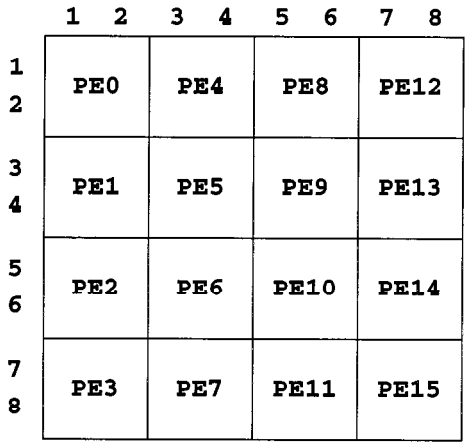
6.2 Geometry and Weights

The concept of *geometry* in this model is an abstraction of the dimensional distribution that simplifies the maintenance and declaration of several arrays with similar dimensional distributions. Change a single **GEOMETRY** declaration, and the distribution of all arrays distributed according to that geometry are changed automatically when the program is recompiled. It is similar, in some regards, to the **typedef** declaration in C. The **GEOMETRY** directive is also retained in the CRAFT-90 implementation.

The following example demonstrates how to declare a geometry.

```
CDIR$ GEOMETRY G(:BLOCK, :BLOCK)
      REAL A(8,8), B(8,8)
CDIR$ SHARED (G) :: A, B
```

The declaration of the geometry **G** declares a distribution that can be used to distribute multiple arrays in the same way. In this example, the arrays **A** and **B** are declared to have the distribution specified by **G**. The following figure shows the distribution for these arrays across 16 PEs.



The CRAFT-77 implementation also supports the concept of *weights* that allows some control over how many processors are assigned to each distributed dimension. A weight is an integer assigned to a distribution that can be used as a PE ratio. For example:

```
REAL C(4,8)
CDIR$ SHARED C(l:BLOCK, 2:BLOCK)
```

recommends that twice as many PEs be assigned to the second dimension as are assigned to the first. The weight specifiers are not being carried over to CRAFT-90 because they are not used very often in CRAFT-77, and sometimes confuse users when this "recommendation" was not honored by the compiler. By default, the compiler will determine how many PEs are assigned to each distributed dimension. CRI will explore providing a different mechanism that allows the user finer control over how processors are assigned to array dimensions.

6.3 Data Mapping Functions

Several data mapping functions are supported in CRAFT-77 that provide the user with low level access to the distribution mechanism assigned to a shared array. The data mapping intrinsics are: **BLKCT** which returns the number of blocks of elements assigned to a PE, **LOWIDX** which returns the lowest index for a particular block, **HIIDX** which returns the highest index for a particular block, **HOME** which returns an integer that represents the processor on which a shared element resides, and PES which returns an integer that represents the number of PEs assigned to a particular dimension. The data mapping intrinsics are also supported in CRAFT-90, because low level access to shared arrays helps blend the implicit programming style with the explicit styles.

7 Subroutines

The CRAFT-77 implementation is based on the principal that the efficiency of the code generated in a routine is dependent on the available information, and the programmer is allowed to specify different amounts of information. Additional directives and semantics were implemented in CRAFT-77 to make subroutine invocations more general. Much of this generality is not supported in CRAFT-90.

There are additional language features in Fortran 90 not present in CRAFT-77 (e.g., allocatable arrays, **POINTER** attribute, and derived types). The relationship between these new features and the CRAFT-90 definition requires further study. Furthermore, Fortran 90 introduces the notion of explicit interfaces for subroutines. Explicit interfaces expose the declarations of dummy arguments to the compiler for use at call sites, and are required when certain types of arguments are passed (e.g., assumed-shaped arrays). The final definition of CRAFT-90 will also use explicit interfaces when some types of shared arguments are passed.

7.1 Unknown Distributions

With CRAFT-77, a subroutine's dummy arguments can be declared to be shared, private, or *unknown.* When the declaration is shared or private, the declaration is used to generate more efficient code for references to those arguments.

However, a dummy argument may have its distribution declared as being unknown with the following directive:

CDIR$ UNKNOWN $arg_1, arg_2, .., arg_n$

The motivation for the **UNKNOWN** distribution was to allow the definition of subroutines that accept both shared and private arguments. This makes library interfaces easier to specify. Similarly, a dummy argument can be declared as both unknown and shared with the following directive:

CDIR$ UNKNOWN_SHARED $arg_1, arg_2, .., arg_n$

Dummy arguments that are both unknown and shared accept a shared argument, but the distribution is not known. Again, the motivation was to simplify the specification of library interfaces.

The disadvantage of unknown distributions is unacceptably slow references to these arguments. The increased overhead is due primarily to the general address calculation used (involves significantly more instructions), and the adverse impact on locality analysis. Locality analysis suffers from the absence of available distribution information. Finally, additional overhead has been added to subroutine calling sequences to support these unknown distributions. This overhead had some affect on all routines because a Shared Data Descriptor (SDD) is passed to describe the shared data. This unacceptable overhead results in very little usage of the unknown distributions, but the effects from passing around the SDD are still there. It does not appear worthwhile for CRAFT-90 to support either the SDD or the

unknown distributions. This simplification of the argument passing mechanism also permits a more efficient entry and exit sequence.

Finally, the **IS_SHARED** intrinsic function is present in CRAFT-77 just to check unknown arguments. This intrinsic is, therefore, not needed in CRAFT-90.

7.2 Implicit Redistribution

With CRAFT-77, shared dummy arguments may not match the exact distribution of the corresponding actual arguments passed into a routine. In this case the CRAFT-77 semantics dictated that shared arguments be redistributed to the specified distribution of the dummy argument upon entry, and redistributed back to their original distribution upon exit. These implicit redistribution semantics may cause excessive overhead, and are rarely exploited by programs They are not planned to be carried over to the CRAFT-90 implementation.

Additional language features called *trusted arguments* were implemented in CRAFT-77 to eliminate much of this unwanted overhead. Since implicit redistribution semantics are disappearing, the trusted arguments are no longer needed.

7.3 PE_RESIDENT Arguments

One additional attribute CRAFT-77 supports for dummy arguments is the concept of a shared argument always being accessed locally. An argument that is specified on a **PE_RESIDENT** directive is an assertion by the programmer that all references to this argument are local private references. Undefined behavior occurs if the assertion is not adhered to by the user.

The **PE_RESIDENT** directive is not being carried over to CRAFT-90 because it only applies to dummy arguments. All too often users wanted the assertion to apply to a particular loop instead of an entire routine. Instead, the directive is being replaced with a loop level feature that gives users finer control over when and where shared arrays are referenced locally (see **Shared Loops** for more information).

7.4 Shared-to-Private Coercion

When you pass a shared array into a routine that declares the corresponding dummy arguments as **PE_PRIVATE,** you are, in effect, passing only those values which reside on the current PE. This is called *shared-to-private coercion.* This means that the called routine must declare the array to be the same size as the number of elements that reside on that PE. For example:

```
REAL D(2048,4000)
CDIR$ SHARED D(:BLOCK,:)
```

If array **D** is passed to a routine that declares the corresponding dummy argument as **PE_PRIVATE** then each PE sees an array of the shape:

```
REAL D_LOC(2048/N$PES,4000)
CDIR$ PE_PRIVATE D_LOC
```

and each PE can reference the data as a local private array.

Shared-to-private coercion provides an easy way to mix both CRAFT and explicit communication in the same program. With shared-to-private coercion a distributed array can be operated on locally, with communication occurring through explicit library interfaces. This useful feature is retained in CRAFT-90 and the elimination of the SDD simplifies the conversion to a local address. The ability to mix high level implicit communication language features with low level explicit communication gives users the ability to tune performance critical areas of a program with the fine control available through these explicit communication interfaces.

7.5 Parallel Execution

CRAFT programs initially begin executing in a parallel region on every PE (same as a message passing program). The program remains in a parallel region (with all tasks executing) until a special directive is encountered that delineates a sequential region. The syntax for this directive is:

```
CDIR$ MASTER
```

When this directive is encountered, all tasks, except the master task, wait at the end of the sequential region for the master task to finish executing the executable statements within the sequential region. The end of a sequential region is delineated by the following directive:

```
CDIR$ END MASTER
```

Every program unit that contains a **MASTER** directive must also contain a matching **END MASTER** directive. The **MASTER** and **END MASTER** directives are used extensively and are being retained in CRAFT-90.

8 Work Sharing

Work sharing is achieved by executing a *shared loop* or *array syntax* statements that reference shared data within a parallel region. The CRI MPP Fortran Programming Model defines both *shared* and *private loops.* Both shared and private loops are retained in CRAFT-90.

8.1 Private Loops

Private loops are executed in their entirety by any task that invokes them. No work is shared with other 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 induction variables (which should be declared **PE_PRIVATE)** retain the behavior they have in sequential Fortran programs. No special syntax is required to specify a private loop; it is the default.

8.2 Shared Loops

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 itera-

tions are executed. The lack of a defined ordering allows the implementation to execute iterations concurrently.

One form of shared loop uses the **DOSHARED** directive to indicate the loop can execute in parallel. It supports an *aligned-on* mechanism that places iterations on the processor where elements for that iteration are stored, thereby increasing locality. The programmer is responsible for determining and specifying the proper alignment.

The syntax for the aligned-on distribution directive is:

CDIR$ DOSHARED (Il,I2,...,In) ON *array-ref*

The *array-ref* must be a subscript expression involving the specified loop control variables **(Il, I2,** etc.) and a shared array.

The proper choice of an iteration alignment can often provide a high degree of locality when references in the iteration are close together. The aligned-on 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 following loop:

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

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

The **DOSHARED** loop is being enhanced to permit a specification of which arrays are guaranteed to be local references within the loop. During the execution of the loop, all references to these arrays, for a particular iteration, are made by the PE executing that iteration. The exact syntax has not been finalized. This capability was requested by many users, and is replacing the **PE_RESIDENT** directive.

9 Array Syntax

A subset of the Fortran 90 array syntax notation is supported in CRAFT-77. Array assignment statements are highly parallel operations. Unlike **DOSHARED** loops, their iteration distribution is controlled completely by the compiler. The implementation chooses the iteration distribution that exercises the greatest locality in its execution. For example, given the declaration:

```
CDIR$ GEOMETRY G(:BLOCK)
      REAL A(100), B(100), C(100)
CDIR$ SHARED (G) :: A, B, C
```

The array syntax assignment

```
A = B + C
```

is equivalent to:

```
CDIR$ DOSHARED (I) dist
      DO I=1,100
            A(I) = B(I) + C(I)
      ENDDO
```

where *dist is* a loop distribution mechanism chosen by the compiler.

Several array intrinsic functions that are part of the Fortran 90 standard are supported in CRAFT-77, and retained in CRAFT-90. Some of the more commonly used intrinsics are: **ALL, ANY, COUNT, CSHIFT, DOT_PRODUCT, EOSHIFT, -MATMUL, MAXLOC, MAXVAL, MINLOC, MINVAL, PRODUCT, SUM,** and **TRANSPOSE.** CRAFT-77 also supports the **WHERE** statement with shared arrays, and is retained in CRAFT-90.

10 Synchronization Primitives

CRAFT-90 supports the same set of synchronization mechanisms as CRAFT-77. These include *barriers, locks, critical regions,* and *events.*

Barriers are a fast way of synchronizing all tasks at once. They are implicitly included at the end of every shared loop (including array syntax statements involving shared arrays) and after allocating shared automatic arrays. They can be explicitly included anywhere in a program with the syntax:

CDIR$ BARRIER

Critical sections (a.k.a., *critical regions* and *guarded regions)* are a specialized form of lock that do not require the use of some convention to insure 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 region. The syntax for a critical section is:

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

Every **CRITICAL** directive must have a properly nested matching **END CRITICAL** directive within the same program unit.

Vector updates are assignment statements that modify an array reference which 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 (i.e., indices) that occur more than once. When this is the case, executing the update in parallel can cause race conditions. The Fortran 90 Standard does not address this problem with array syntax statements such as:

X(IX) = X(IX) + V

except to say that it is an error if any of the elements of the index vector **IX** have the same value. Therefore, a directive is supported that directs the compiler to ensure that multiple

updates to a single shared element occur atomically. The vector update executes as parallel as possible otherwise. The syntax is:

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

The directive only applies to the assignment statement immediately following the directive and the assignment must be to a shared variable, including scalars.

Locks are a basic and primitive synchronization method that are generally used to serialize access to some piece of data. Locks that are set to zero are initialized as unlocked. Lock operations are supported by three routines as follows:

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

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 are not easily used to completely serialize access to data. Events are supported by four routines as follows:

CALL SET_EVENT(*[event]*)
CALL WAIT_EVENT(*[event]*)
CALL CLEAR_EVENT(*[event]*)
L = TEST_EVENT(*[event]*)

The argument to the event routines is optional. If no argument is supplied then a hardware mechanism called the *eureka* is used for event communication.

11 Conclusions

There are multiple forces that apply pressure to the definition of a programming model for a massively parallel system. Some programmers want very high level language features, others want very low level primitives that allow them to control every aspect of the program's execution. Some want a very MIMD model with extensive synchronizations, others want the implementation to "figure it all out" for them. The CRAFT-77 model is feature rich, because of this divergence of opinions in the user community over what is the best way to program massively parallel systems. Using the experiences gained from CRAFT-77 on the Cray T3D, CRI is defining an enhanced version, CRAFT-90, that is based on CF90 and will be released on the Cray T3E system. The goal for CRAFT-90 is a more usable language for which good performing code can be generated.

CF77, CF90, Cray T3D, and Cray T3E are trademarks of Cray Research, Inc.

References

1. D. Pase, T. MacDonald, A. Meltzer, *MPP Fortran Programming Model*, Cray Research, Inc., Eagan Minnesota, 1994.
2. *Cray MPP Fortran Reference Manual*, SR-2504 6.2, Cray Research, Inc., Eagan, Minnesota, 1994.
3. *CF90 Fortran Language Reference Manual*, SR-3092 1.0, Cray Research, Inc., Eagan, Minnesota, 1994.
4. *CM Fortran Reference Manual*, Thinking Machines Corporation, Cambridge, Massachusetts, 1991.
5. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. W. Tseng, and M. Y. Wu, *Fortran D Language Specification*, Rice University, Houston, Texas, 1991.
6. P. Mehrotra and J. Van Rosendale, *Programming Distributed Memory Architectures Using Kali* ICASE, NASA Langley Research Center, Hampton, Virginia, October, 1990.
7. F. Thomson Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufman Publishers, San Mateo, California, 1992.
8. B. Chapman, P. Mehrotra, and H. Zima, *Vienna Fortran - A Fortran Language Extension for Distributed Memory Multiprocessors*, ICASE, NASA Langley Research Center, Hampton, Virginia, 1991.
9. High Performance Fortran Forum, *High performance Fortran Language Specification*, Ver. 1.0, May 3, 1993
10. Constantine D. Polychronopoulos, *Parallel Programming and Compilers*, Kluwer Academic Publishers, Norwell, Massachusettes, 1988.
11. Hans Zima with Barbara Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press Frontier Series, New York, New York, 1991.