MPP Fortran Programming Model

Douglas M. Pase, Tom MacDonald, Andrew Meltzer

Cray Research, Inc.
655F Lone Oak Drive
Eagan, Minnesota 55121

This report describes the initial MPP Fortran programming model which will be supported on the first phase MPP systems. Based on existing and proposed standards, it is a work sharing model which combines features from existing models in a way that is both efficiently implementable and useful.

The work represented by this paper was supported by
The Defense Advanced Research Projects Agency
under Agreement No. MDA972-92-H-0002
dated 21 January, 1992

1.0 Overview

Many programming models for massively parallel machines exist, and each has its advantages and disadvantages. In this report we present a programming model which attempts to combine features from other programming models which 1) can be efficiently implemented on our massively parallel processor (MPP) machine, and 2) are useful in constructing highly parallel programs. It is a *work sharing* model rather than a *fork-join* model. A work sharing model distributes parallel work across existing tasks, while fork-join models create new tasks to execute parallel work. 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* 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.

The data distribution declarations were adapted in part from Rice University's Fortran-D project. The work distribution directives are adapted from a wide variety of sources, including Cray Autotasking™ and workshops on parallel programming held at the University of Illinois. Concepts embraced in this model can be found in many other sources as well.

2.0 Introduction

The MPP programming model is an attempt to allow the user a range of control over the MPP hardware. This range extends from a low level 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 allows users to write programs which 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 which specifies what the system as a whole will compute, or 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 and placement of data, parallel execution, the distribution of loop iterations, synchronization primitives, distributed and sequential I/O, subroutine interfaces, and special intrinsic functions that support parallel reductions, parallel prefix operations, and segmented scan operations. Additional intrinsic functions allow the user to access low level detail about array distributions as well as more abstract concepts like the virtual machine topology used to distribute one or more arrays.

This programming model distinguishes between data objects that are private to a task (`PE_PRIVATE`) and those that are shared among all tasks (there is a one to one relationship between tasks and PEs.) 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 resides entirely on only one 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. 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.

Programs initially execute as a single task on a single PE, and parallelism is initiated explicitly by the user. Because of the number of PEs involved, it is expected that users will have either very short sequential regions at the very beginning of a program (for data initialization), or that the first statement executed will invoke parallel execution. Long sequential segments anywhere within a program would result in highly inefficient machine utilization. Tasks are not created dynamically in the sense that UNIX® processes are, but rather one task is created per processor and parked until parallel execution is invoked. Each task also has an identity which it can use to distinguish itself from other tasks.

Loops do not create parallelism in a program. Rather they may execute serially, one loop in each task, or they may “share” the work, distributing it across all available processors. Distributed loops (called *shared loops* in this model) are work sharing constructs rather than task creating constructs. A rich set of iteration distribution mechanisms is provided in order to support highly efficient execution under a variety of conditions.

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, that is, 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. It is anticipated that barriers, locks, and critical sections will be supported by extremely fast hardware mechanisms. This hardware support will greatly reduce the impact on performance that using these primitives has had on other systems.

Input and output also have distributed and non-distributed versions. Distributed I/O statements (called *shared I/O*) allow the user to specify what all tasks will read or write as a whole, and allow the system to distribute the work as it sees fit. This works in the same way as shared loops in that the user specifies what work is to be done by all tasks and allows the system to divide up the work among the tasks. Sequential I/O is also supported, and it allows the user to specify the work to be done by each individual task. Sequential I/O (called *private I/O*) is similar in concept to local loops in that each task must do all of the work specified by the statement.

Subroutine interfaces are extended to accomodate distributed data. While it is tempting to require that the distribution attributes of arguments in function calls exactly match the corresponding formal arguments in function definitions, it is perceived that such a restriction would cause 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 dumy 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 different distributions are specified, redistribution is done implicitely by the compiler.

Two categories of intrinsic functions are supported in this model: high level reduction and reduction-like functions and low level functions that give information about array distributions. The high level functions perform several similar computations. They perform reduction operations, parallel prefix operations, and segmented scan operations. These three types of operations work in general for any associative binary operator. The low level intrinsic functions provide usable information about how an array is distributed across the machine.

The model supports message passing primitives based on the PVM model (Parallel Virtual Machine.) It 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 allow the codes written using this programming model to run correctly on machines that do not support the directives. Code written using this model produces mathematically identical results (modulo hardware limitation considerations) on a sequential machine if the directives are ignored so long as there is no non-deterministic behavior in the user's program and the program does not use any of the MPP specific intrinsic functions.

3.0 Data Objects

A data object is any program data storage area, whether it is 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`). The first set of attributes are *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 are called *shared*, because such objects are accessible to all tasks.

3.1 Private Objects

Private data storage is replicated. Each declaration of a private array or scalar object causes one such object with the specified name to be created for each task that executes.

Private data is not distributed. An object is always allocated completely on 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. Data can be specified to be private using the PE_PRIVATE directive.

The default distribution attribute is PE_PRIVATE–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 var_1, var_2,..., var_n

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, and automatic arrays.

3.2 Shared Objects

Shared data objects are accessible to all tasks. Only one data object exists for each declaration of a shared data object. Scalar variables are always allocated on a single PE, although not all shared scalars are necessarily allocated on the same PE. Blank common blocks may not be shared, nor may objects in blank common be shared, but objects local to a subroutine, including automatic arrays, can be shared. Character data may not be shared. A shared object is considered *distributed* across the program's PEs.

Shared data objects' distributions fall generally into two categories: canonical distributions and dimensional distributions. Canonical distributions spread the elements across the machine independently of the dimensionality (or rank) of the object being distributed. The first few elements are allocated on PE 0, the second few elements are allocated on PE 1, etc., until a block of elements is allocated on the last PE. The allocation then "wraps around," putting the next block on PE 0, then PE 1, etc. This process continues until all elements in the object are allocated. The block size used in the distribution is implementation dependent and may be different from one generation of machine to the next. Multidimensional array references are first linearized before distribution, and canonically distributed common blocks retain their storage associations as defined in Fortran.

Dimensional distributions may be applied to any shared array. They may *not* be applied to scalar objects or 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 4 processors mapped to each dimension. (This works because $4^3 = 64$.) Alternatively it might have 8 processors mapped to the first dimension, 4 mapped to the second, and 2 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 would be chosen if the first dimension weight were 4 (because it is 4 times larger than the last dimension), the second dimension weight were 2, and the last dimension weight were 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 task receives one contiguous block of elements. The second designation, `BLOCK(M)` indicates that each processor 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 were to be allocated over 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*. 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 used for 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 weightings 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 `X(64)`. Suppose that this array is to be distributed over 4 processors. If the array is distributed `BLOCK`, then 64/4 = 16 array elements in a contiguous block of indices is allocated to each processor. 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 `Y(16,16)` and that it is mapped to an 8 processor machine. Furthermore, suppose that the first dimension is assigned 4 processors, the second dimension is assigned 2, 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 numbers of PEs assigned to each dimension are used to determine the exact PE number. The PE number is given by $0+1\times4$, 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.

Figure 1. Memory Allocation Pattern for Y

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)

Individually distributed arrays and common blocks always have their first element on PE 0, regardless of the number of elements or the particular distribution it uses. This allows a user to align distributed arrays in a primitive but efficient fashion in order to guarantee that when array references are local to one task 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, four arrays `A(M,N)`, `B(M,N)`, `C(M,N)`, and `D(K,L)`, where `K`, `L`, `M`, and `N` are all distinct, are distributed `A(:BLOCK,:BLOCK)`, `B(:BLOCK, :BLOCK)`, `C(:BLOCK,:)`, and `D(:BLOCK,:BLOCK)`. All arrays have the same rank, that is, they all have 2 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 `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.

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. Canonical arrays are padded in this way such that their total sizes are allocated in integer multiples of one canonical block times the number of available processors.

Dimensionally distributed arrays have more severe constraints on their allocations than do canonically distributed arrays. 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 compiler internally converts each dimension to the next higher integer power of two; it also converts block sizes and the number of processors assigned to each dimension to the next higher integer power of two.

The syntax for declaring a shared data object is

CDIR$ SHARED var_1, var_2, ..., var_n

where var_i represents the name of a data object, which may be the name of a scalar variable, an array, or a named common block enclosed in slash marks (“`/`”). The above syntax is adequate to declare shared scalars and shared objects that are canonically distributed. To distribute an array by dimension (neither scalars nor common blocks can be distributed dimensionally) one need only append a description of each dimension’s distribution, with any desired weights, to the array name. This generally has the appearance $var_i(\alpha_1, \alpha_2, ..., \alpha_r)$ where `r` is the rank of the array. The notation α_j represents a keyword from the selection: `:BLOCK` or `:BLOCK(M)`, optionally with weights, or “`:`”. Weighted dimensions are represented by `w:BLOCK` or `w:BLOCK(M)`, where `w` and `M` are integer expressions.

When distributing an array that is either 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. An example of the declaration of a canonically distributed common block might be

```
      COMMON /XXX/ A, B, C
CDIR$ SHARED /XXX/
```

A canonically distributed array might be declared as

```
      REAL X(1000)
CDIR$ SHARED X
```

Dimensionally distributed array declarations look like

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

Initial values for shared data objects are undefined, but shared 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 shared attributes in this model. Primarily this means all shared objects may be `DATA` initialized *except* those that occur in blank or unnamed `COMMON`, dummy arguments, and automatic arrays.

3.3 Geometry

The concept of *geometry* in this model is an abstraction of the dimensional distribution that gives two fairly diverse advantages. The first is that 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. It is similar in some regard to the `typedef` declaration in C. The second advantage is that it creates an abstract object that can be used to specify and determine the relationship between tasks. For example, a three-dimensional geometry represents a three-dimensional torus (or mesh, depending on how it is used) with exactly as many nodes in the torus as there are tasks available. A geometry can be used to determine tasks that are neighbors in a computation. The information on neighboring tasks is available through the `GEOMMAP` and `TASKMAP` intrinsic functions described in Section 9.0.

The syntax for declaring a geometry 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 $geom_1(\alpha_1, \alpha_2, ..., \alpha_r)$, ...
CDIR\$ SHARED (*geom*) [::] var_1, var_2, ..., var_n

Here, α_i has the same meaning as it has in Section 3.2, 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(13,9)
CDIR$ SHARED (G) :: A, B
```

The declaration of `G` describes a distribution which is then applied to arrays `A` and `B`.

3.4 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 in order to maximize reference locality. 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.

Arrays may also be redistributed across subroutine boundaries. A dummy argument which is distributed differently than its actual argument in the calling routine is automatically redistributed upon entry to the subroutine by the compiler, and automatically redistributed back to its original distribution at the subroutine exits. If the distributions are identical or the `UNKNOWN` or `UNKNOWN_SHARED` directive is used (Section 8.1.1), no redistribution occurs.

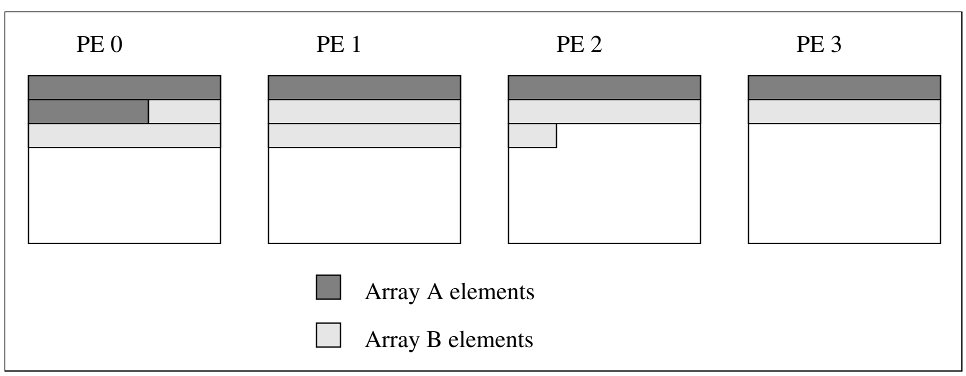
3.5 Common Blocks

As mentioned before, whole named common blocks can be distributed canonically. Neither unnamed common blocks nor objects within unnamed common blocks may be shared. Data objects within a common block may also be distributed, but only if the common block itself is not distributed. It is *not* the same thing to say that a common block is shared as it is to say the objects within a common block are shared. Figure 2 shows an example of this difference.

Figure 2. Shared Common Block

Given the common block declaration below, the following data distribution across a 4 PE machine will occur.

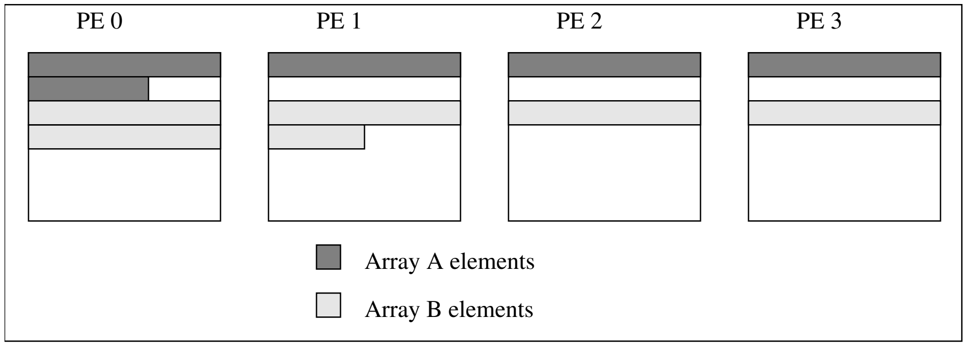
```
      COMMON /XXX/ A(140), B(180)
CDIR$ SHARED /XXX/
```



Contrast the above example with the canonical distribution declaration for each object in the common block in Figure 3.

Figure 3. Shared Objects within a Common Block

```
      COMMON /XXX/ A(140), B(180)
CDIR$ SHARED A, B
```



Data objects within a common block that are individually shared lose the storage association that is implied by being part of a common block. Semantically it is as if the objects were placed in completely separate common blocks. The first element of each shared object has its first element allocated on PE 0, but scalars can be allocated anywhere within the machine.

Because the storage association is lost when arrays are individually distributed, it is of questionable meaning to declare a named common in one subroutine with separate distributions for objects in the common, and have the same named common in another subroutine with a different list of objects and distributions.

Shared common blocks retain the storage association implied by Fortran. Thus a shared common block in one subroutine is allowed to have a different list of objects when declared in a different subroutine. For example, the program fragment

```
      SUBROUTINE S1()
      COMMON /CB/ A(100), B(100,100)
CDIR$ SHARED /CB/
      ...

      SUBROUTINE S2()
      COMMON /CB/ C(100,100), D(100)
CDIR$ SHARED /CB/
...
```

retains the same storage associations (between the various objects declared in the common block) in this model as they are defined in Fortran. In this example the array `A` is associated with the first column of `C`, the last column of `B` is associated with `D`, `A(101)` is equivalent to `B(1,1)` and `C(2,1)`, etc. Note that this would not be true if `A`, `B`, `C`, and `D` were individually distributed.

Objects within the common block do not have their allocations padded to fill a constant block size on all PEs, although the common block itself may be padded to use the same amount of memory on each PE. No alignment guarantees are made, except that the first element of the first object in the common block (`A(1)` and `C(1,1)`) will be located on

PE 0. All objects within a distributed common are distributed canonically, but they are not necessarily aligned on canonical distribution block boundaries.

4.0 Parallel Execution

The MPP Fortran programming model is built primarily around the notion of work sharing. Constructs within this model provide access to mechanisms that distribute work among the available executing tasks. Shared data is 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.

4.1 Parallel Sections

Programs initially begin executing in a sequential region. Only one task on one PE, PE 0, executes initially. The program remains in that execution mode until it encounters a special directive to execute parallel tasks. The syntax for this directive is

```
CDIR$ PARALLEL ALL
```

The program then continues to execute all tasks in parallel until the directive

```
CDIR$ END PARALLEL
```

is encountered. Every program unit which contains a `PARALLEL` directive must also contain a properly nested matching `END PARALLEL` directive. `END PARALLEL` directives carry an implicit barrier synchronization. The `PARALLEL ALL` directive may also optionally contain a `COPY(`var_1`,...)` directive. This directs the contents of the private data for var_1, etc. on PE0 to be broadcast to all of the other PEs, basically ensuring that they all start with identical values for each copied data item. The syntax for a `PARALLEL ALL` with a `COPY` is

```
CDIR$ PARALLEL ALL, COPY(var1, var2, ...)
```

All tasks are created upon program start-up 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, it is the task that executes. While executing in a sequential region all other tasks are parked at a barrier. This mechanism allows tasks to be parked and unparked rapidly. However, because of the number of processors involved, 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.

Parallel regions may be nested but the effect is that the inner directive is ignored. Thus if a subroutine must be executed in a parallel region it is safe to place parallel directives around the appropriate code segments. If the subroutine call takes place within a sequential region, the subroutine will execute in parallel when the `PARALLEL` directive is encountered. If the subroutine call takes place within a parallel region, the subroutine's `PARALLEL` and `END PARALLEL` directives will be ignored. Note that encountering an `END PARALLEL` directive does not guarantee that the program will execute in a sequential region, unless it is the outermost `END PARALLEL` directive. To otherwise guarantee sequential execution one must use code similar to

```
CDIR$ BARRIER
      IF (MY_PE().EQ.0) THEN
         ... (serial code) ...
      END IF
CDIR$ BARRIER
```

which will always work whether the code is executing sequentially or in parallel. The MY_PE() intrinsic is defined below in Section 4.2.

4.2 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 constant called N$PES which gives the total number of CPUs (as well as the number of tasks) available to the program. It may be used anywhere a named constant may appear. It has the same value in either a sequential or parallel region.

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. 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.

The task topology defined by the intrinsic function MY_PE() defines a one-dimensional torus (or mesh). Wrap around is achieved by treating the operations as operations in a linear congruence, that is, by doing a "mod N$PES" operation to any manipulation within the space. For example, a task may find one of its neighbors in the torus by evaluating the expression (MY_PE()+1) mod N$PES.

Model support is not restricted to a one-dimensional torus. However, support is currently planned only for regular mesh and torus topologies, although our model design does not completely preclude supporting irregular topologies at some future date. Support for higher-dimension tori comes through the GEOMETRY statement described in Section 3.3 and the map functions defined in Section 9.2. Further discussion of multidimensional topologies is deferred to the sections mentioned.

5.0 Loops

As with data objects, loops also 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 declare a loop as private--it is the default.

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, nor in most cases the PE on which they will be executed. The lack of a defined ordering allows the system to execute iterations concurrently.

Shared loops may use either a pre-scheduling or a self-scheduling iteration distribution mechanism. Pre-scheduling is a low overhead mechanism that assigns iterations to tasks prior to executing the first iteration. The iteration distribution mechanism is itself completely distributed and requires no communication between tasks.

Pre-scheduled loops are not adaptive, so large variations in iteration execution times can cause load imbalances among the tasks. When the `UNIFORM` iteration distribution mechanism is used, the iteration sequence can be randomized (described in Section 5.3), which in some cases can ameliorate the problem.

Self-scheduled loops are adaptive, that is, iteration blocks are distributed to idle tasks based on conditions that occur at run time. Tasks that are busy do not obtain more work until they become idle. In this way the distribution mechanism adapts its behavior to the execution behavior of the loop. Because a shared loop is one loop for which iterations are executed concurrently, induction variables do not function in the same way that they do for sequential Fortran programs. Furthermore, no guarantees are made that iterations will have local references--locality occurs only by coincidence. Self-scheduled loops are also randomizable.

Self-scheduled loops use a very fast distribution mechanism that is at least partially (and may be wholly) centralized, but because it is centralized there can be a lot of contention for the mechanism. For this reason these mechanisms will be most effective in situations where the iteration executions have very large variance in the iteration execution times, or where the iteration blocks contain a large quantity of work.

The general syntax for shared loops is as follows:

```
CDIR$ DOSHARED (I1,I2,...,In) mechanism
      DO I1 = L1, U1, S1
         DO I2 = L2, U2, S2
            ...
               DO In = Ln, Un, Sn
                  ...
               END DO
            ...
         END DO
      END DO
```

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 within another shared loop, the compiler will issue a warning or syntax error. If an iteration of a shared loop calls a subroutine that also contains a shared loop, the inner shared loop is executed as a private loop. In the directive, *mechanism* is the specific iteration distribution mechanism the user desires, described below.

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 they have different behaviors. Only program performance depends on what mechanism is used to distribute a shared loop. In addition, private loops and shared loops function identically in sequential sections.

5.1 Pre-Scheduled Shared Loops

This programming model supports two styles of pre-scheduled loops: uniform distributions and aligned distributions. The uniform mechanism spreads the iterations uniformly across the available tasks without regard to data locality. The aligned mechanism places each iteration on the processor where the element for that iteration is stored, increasing locality. The user is responsible for determining and specifying the proper alignment.

5.1.1 Uniform Iteration Distribution

The uniform iteration distribution mechanism is a low overhead form of block scheduling. No reference locality is guaranteed by this mechanism. Because it is not adaptive its performance may suffer badly when large iteration execution variation occurs. However, distribution overhead is so much lower than adaptive mechanisms that execution variation must be quite high before an adaptive loop is more efficient, and randomization of loop iterations (Section 5.3)can further improve their performance.

Syntax for the uniform distribution mechanism directive is

$$\texttt{CDIR\$ DOSHARED (}\mathtt{I}_1,\mathtt{I}_2,\ldots,\mathtt{I}_n\texttt{) UNIFORM}$$

Iteration randomization occurs when the directive

$$\texttt{CDIR\$ DOSHARED (}\mathtt{I}_1,\mathtt{I}_2,\ldots,\mathtt{I}_n\texttt{) RANDOM UNIFORM}$$

is used.

5.1.2 Aligned Iteration Distribution

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. Since Y(I) resides on the same PE, all references are completely local.

The general syntax for aligned loops is

$$\texttt{CDIR\$ DOSHARED (}\mathtt{I}_1,\mathtt{I}_2,\ldots,\mathtt{I}_n\texttt{) ON X(}f_1(\bar{\mathtt{I}}),f_2(\bar{\mathtt{I}}),\ldots,f_r(\bar{\mathtt{I}})\texttt{)}$$

where n is the number of shared loops in the nest, r is the rank of the array X, and f_i is of the form $f_i(\bar{\mathtt{I}}) = a_{1i}I_1+a_{2i}I_2+...+a_{ni}I_n+b_i$. Since f_i represents an index expression, all a_{ji} and b_i must be integers.

Ultimately it is our intention to support a general form of aligned distribution, with arbitrary integer linear expressions as defined above. This is a well defined problem and we see an efficient solution, but the solution is complex so we will implement only the most useful cases at first. Initially we will support index expressions of the form `aI+b`, where the loop control variable `I` is used in at most one array index expression. Other forms will be implemented as time permits.

5.2 Self-Scheduled Shared Loops

Three self-scheduled iteration distribution mechanisms are supported in this programming model. Two of them are block scheduling mechanisms, which differ only in how they specify the block size. The third is a mechanism in which the iteration block sizes decrease with the amount of work available, similar to Guided Self-Scheduling. All three divide the iteration space into blocks of iterations and distribute them as tasks indicate they are required.

5.2.1 Block Iteration Distribution

Adaptive block iteration distribution comes in two flavors. The first specifies the size of each block or chunk, the second specifies the number of chunks the system is to create. The two directives have the syntax

CDIR$ DOSHARED (I_1,I_2,...,I_n) *mechanism*(M)

`M` and `N` may be any integer expression, evaluatable at run time. The *mechanism* is either `CHUNKSIZE` or `NUMCHUNKS`. However, the program gives predictable results only when all tasks reduce the expression to the same value. The syntax for randomizing the iterations is

CDIR$ DOSHARED (I_1,I_2,...,I_n) RANDOM *mechanism*(M)

5.2.2 Guided Iteration Distribution

Block scheduling has been shown to be inefficient for evaluating certain loops. The `GUIDED` mechanism initially distributes large blocks of iterations and gradually decreases the block size until none remain. Thus if one task receives an unusually slow block of iterations the system has greater flexibility in distributing the remaining work. In particular, as the block sizes are reduced the probability that an iteration block has an unusually large amount of work is also reduced.

The guided iteration distribution mechanism is modeled after Guided Self-Scheduling (GSS), but GSS is perceived to have certain deficiencies for an MPP environment. This mechanism attempts to remedy those concerns.

The syntax for the guided iteration distribution mechanism directive is

CDIR$ DOSHARED (I_1,I_2,...,I_n) GUIDED(M)

The syntax for randomizing the iterations with this mechanism is

CDIR$ DOSHARED (I_1,I_2,...,I_n) RANDOM GUIDED(M)

5.3 Iteration Block Randomization

When the `RANDOM` iteration distribution modifier is specified on uniform, block, or guided loop distributions the iterations selected for each task are non-contiguous in the iteration space. The non-contiguous iteration sets break up any pattern that might exist in the iteration work distribution (such as occurs in triangular loops). The nature of linear congruences insures that the randomization process is inexpensive and that all iterations are assigned to exactly one block of iterations. If the work distribution has no pattern, the randomization has no effect.

5.4 Array Assignment

Array syntax is supported in this programming model. Array assignment statements are highly parallel operations. 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 which a user is not able to do with `DOSHARED` loops. For a detailed explanation of the semantics of mixing private and shared variables, see Section 9.1.4.

For example, given the declaration:

```
CDIR$ SHARED A, B, C
      DIMENSION A(100), B(100), C(100)
```

When `A` is `SHARED`, the array syntax assignment `A=B+C` is semantically equivalent to

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

where *mechanism* is chosen by the compiler. When `A` is `PE_PRIVATE` it is equivalent to a local loop.

5.5 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 is redistributed and when `SHARED` automatic arrays are allocated. Barriers may be explicitly removed from a loop by placing a directive at the end of the loop. The syntax of that directive is

```
CDIR$ NO BARRIER
```

6.0 Synchronization Primitives

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

6.1 Barriers

Barriers are a fast 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. 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 program units that allocate distributed memory dynamically (discussed in Section 8.0). They may be explicitly included anywhere in a program with the syntax

```
CDIR$ BARRIER
```

Barriers 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 which may be executed sequentially or in parallel, and if executed in parallel require synchronization.

6.2 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 wide 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 insure 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. While one task may only *lock* a lock if the lock is unlocked, any task may *unlock* a lock at any time.

Locks do not require initialization or release functions. Locks that are set to zero are initialized as unlocked. Lock operations are supported by three intrinsic subroutines, which use the syntax

```
CALL LOCKON(lock)
CALL LOCKOFF(lock)
L = LOCKTEST(lock)
```

The subroutine `LOCKON` sets the lock. If the lock is set it spin-waits until the lock is cleared, otherwise the lock is set immediately. `LOCKOFF` clears a lock whether it is set or not. `LOCKTEST` 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.LOCKTEST(LCK)) THEN
   ...(the lock is ours)...
   CALL UNLOCK(LCK)
ELSE
   ...(do something else)...
END IF
```

For all three functions the operand `lock` is expected to be a 64-bit integer. Locks will initially require a 64-bit integer operand, but future support for a 1-bit integer in the compiler will reduce the size of a lock to a single bit.

These spin-locks are designed to efficiently support many locks that infrequently have access collisions. Contention-free access is expected to be inexpensive, but with heavy contention access is potentially both expensive and unfair.

6.3 Critical Sections

Critical sections 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 section. The syntax for a critical section is

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

Every `CRITICAL` directive must have a matching `END CRITICAL` directive within the same program unit. 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. These directives must also be perfectly nested.

6.4 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 intrinsic routines, namely `EVPOST`, `EVWAIT`, `EVTEST`, and `EVCLEAR`. `EVPOST` sets, or posts, an event. It declares the event to have occurred. No prior conditions are imposed upon this routine. Any event can be posted at any time, whether the state of the event is already posted or cleared. `EVWAIT` suspends task execution until a specified event occurs. `EVTEST` returns the state of an event, i.e., whether it is posted or cleared. `EVCLEAR`, of course, clears the event. The syntax for each routine is

```
CALL EVPOST(event)
CALL EVWAIT(event)
CALL EVCLEAR(event)
S = EVTEST(event)
```

As with locks, all events will initially have a 64-bit data type. But later on when the compiler supports 1-bit integers, events will require only a single bit.

6.5 Vector Update Loops

Vector updates pose a difficult problem. A vector update executed in parallel may attempt to update the same element of an array more than once at the same time, causing a race condition and potentially giving incorrect results. The option of computing vector updates sequentially is generally not acceptable, and forcing the programmer to do all of the necessary synchronization is error-prone and unnecessary. This programming model supports a set of vector update primitives for addition, multiplication, maximum, minimum, binary AND, OR, and exclusive OR. The directive `ATOMIC UPDATE` directs the compiler to ensure that multiple updates to a single element occur atomically. The vector update executes as parallel as possible otherwise. The syntax of the `ATOMIC UPDATE` is

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

The `ATOMIC UPDATE` directive only applies to the statement immediately following the directive and may be placed before non-vector updates as well as long as one of the proper operations is being executed.

7.0 Input and Output

7.1 Private I/O

As with memory and loops, input and output may be either private or shared. Each task performs private read and write operations as if it were the only task looking at a file. When conflicts arise between two tasks, with one or both attempting to write to the same file, undefined behavior results just as it would between two UNICOS® processes on a shared memory multiprocessor system like the Cray Y-MP. Private I/O is the default, so 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.

Although private I/O is easy enough to understand in theory, in practice it is difficult to use effectively for reading and writing shared arrays. It requires seeking to a point in a file that represents, because files are linear sequences of bytes or words, a linearization of what may be a multidimensional array.

7.2 Shared I/O

Shared I/O is most intuitive when used to read shared objects from or write them to a file, although it may also be used to read and write private data. All tasks must participate in shared I/O operations, so the I/O operations carry an implicit barrier to execute. Of course all tasks must have the same list of objects to be read or written, too.

Shared I/O allows the programmer to specify the behavior, for the whole machine, of an I/O operation. Thus if a shared array `X(1:1024)` is to be read using shared I/O, however it is done internally, the statement specifies that all processors together are to read 1024 elements and store them in `X`. Depending perhaps on the distribution of `X`, one task reads some subset of the elements of `X`, and another task reads a different subset. If this were specified as a private I/O statement, then *each task* would read 1024 elements.

When `X` is a private array, the operation specifies that all copies of `X` be written or read, in the order specified by the processor number, starting from PE 0 and going through `N$PES`-1. Thus if executed on a 512 PE machine, this example causes 512×1024 words to be written. Scalar variables, whether shared or private, may also be written or read. Shared scalars cause one datum per scalar to be written or read; private scalars cause N$PES values per scalar to be written or read.

The directive for declaring shared I/O is

```
CDIR$ SHARED_IO
```

This directive is placed immediately prior to a shared I/O operation. It must be included on open, close, read, write, and inquire statements. File units opened for private I/O must not be used for shared I/O, just as file units opened for shared I/O must never be used for private I/O.

Private and shared I/O behave in exactly the same way when they are executed in a sequential region.

8.0 Subroutines

Two opposing goals arise in designing the behavior of subroutines. High performance is crucial to the success of the CRI MPP machine, 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 execute a barrier, redistribute data, allocate shared data objects dynamically, and it executes only private loops and I/O. 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 requires all tasks to share the work in some manner. This may come about because the routine creates shared dynamic arrays, explicitly uses a barrier, or uses some work construct that implicitly uses a barrier, such as shared loops or shared I/O.

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, while 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 to occur.

8.1 Data Objects

Fortran 90 and Cray Fortran (cf77) both support several types of data objects within subroutines and functions. A list includes named and unnamed common blocks, dummy arguments, local scalars and arrays, and automatic arrays. For MPP 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 3.5. Other object types are described in the following sections.

8.1.1 Dummy Arguments

Dummy arguments may be declared as `SHARED` (canonically or dimensionally), `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 which apply to this redistribution are

- If the dummy argment 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.
- If the dummy argument is declared `UNKNOWN` or `UNKNOWN_SHARED` then a general addressing scheme is used and no redistribution is done.
- If the dummy argument is declared private and the actual argument does not match exactly, the addressing will assume that the passed parameter matches. 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.
- 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 back to its original form at the subroutine exits.

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

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

When the distribution is unknown at compile time it is determined at run time. The subroutine assumes the most general distribution possible, which causes access to the dummy argument to perhaps have somewhat lower performance. Its advantage is that this mechanism may be used to create fairly general subroutines and functions which accept distributed or private arrays as arguments.

8.1.2 Local Objects

Subroutines may declare local variables, i.e. objects local to a subroutine, that are shared or private. Arrays and scalars both may either be shared as well as private. Data initialization rules for local objects follow the rules for `cf77`, that is, 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 2.0. Subroutines that use shared local objects implicitly contain barriers on entry to and exit from each subroutine.

8.1.3 Automatic Arrays

Both shared and private automatic arrays are supported in this model. Subroutines which allocate shared automatic arrays implicitly contain barriers upon entry to and exit from the routine. 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.

8.2 Parallel Regions

Subroutines may include the declaration of parallel regions. If the subroutine executes within a parallel region, when the `PARALLEL` and `END PARALLEL` directives are encountered they are ignored. If the subroutine is executed in a sequential region, the program immediately begins concurrent execution of all tasks.

8.3 Loops

Program units may use shared loops whether or not they declare a parallel region within the subroutine. Shared loops that execute within a sequential region behave as if they were ordinary loops. Shared loops that execute within a parallel region, whether the parallel region was defined within the scope of the program unit or not, behave as defined in Section 5.0. Routines that use shared loops have barrier synchronization implicitly associated with the loops.

9.0 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 is 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.)

9.1 Data Parallel Functions

The data parallel functions supported consist of three basic function families, namely reduction functions, parallel prefix functions, and segmented scan functions. 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. Parallel prefix functions are useful solving dense linear algebra problems, such as Gaussian elimination without pivoting. Segmented scan functions are a generalization of parallel prefix functions; they offer the ability to perform many prefix operations over sub-ranges of an array. This makes them useful for solving linear algebra problems with some representations of sparse matrixes. Some representations where scan functions are useful are banded, sky line, and block tridiagonal sparse matrices.

9.1.1 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.

```
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`.

9.1.2 Parallel Prefix Functions

Parallel prefix functions behave similarly to reduction functions, but they retain all partial reductions. Let ⊕ be some associative binary operation (for example, 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, 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.

9.1.3 Segmented Scan Functions

Scan functions behave similar 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, 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, innermost dimensions first. When `DIM` is specified, the scan operation occurs only along the specified dimension.

9.1.4 Semantics of Data Parallel Functions

Data parallel functions use the semantic rules as follows:

- When only one processor participates in the data parallel function, it takes on the meaning indicated.
- When multiple processors are involved, the behavior varies slightly depending on the attributes of `RESULT` and `ARRAY`.
- When `RESULT` and `ARRAY` are both shared, `ARRAY` is processed and the value is stored in `RESULT`.
- When `RESULT` is private and `ARRAY` is shared, `ARRAY` is processed and the value is broadcast to all copies of `RESULT` that participated in the function.
- When `RESULT` is a single shared object referenced by all participating processors, `ARRAY` is private, and more than one processor participates in the function, a race condition exists and the value of `RESULT` is undefined. For the reduction functions, when only one processor participates, or each processor 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.
- When `RESULT` and `ARRAY` are both private, the function takes on its sequential semantics and the function is performed locally by each processor participating in the function.

Note that when `ARRAY` is shared, all processors must participate and a barrier synchronization is implied.

9.2 Data Mapping Functions

The data distribution mechanism described in Section 3.0 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.

Seven functions are supported. Three of the functions provide direct access to the blocks of data distributed by the mechanism. Two of the functions provide information about how data is mapped to the processors. The last two functions provide information about the multidimensional virtual torus used to map data to processors. A detailed description of each function follows.

`BLKCT(A,D,P)` returns an integer which represents the number of blocks of elements in the D^{th} dimension of array `A` that are resident on processor `P`.

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

`LOWIDX(A,D,P,K)` returns an integer which represents the lowest index of block `K` in the D^{th} dimension of array `A`, which is resident on processor `P`.

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

`HIIDX(A,D,P,K)` returns an integer which represents the highest index of block `K` in the D^{th} dimension of array `A`, which is resident on processor `P`.

```
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))
      ...
      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 which represents the processor on which `X` resides. `X` is a scalar object rather than an array.

```
      HOME(X)
```

`PES(A,D)` returns an integer which represents the number of processors used in the `D`th dimension of array `AG`; `AG` may also be a geometry.

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

`GEOMMAP` is an array valued function which converts a processor identification number (`UP`) into a coordinate point (or tuple) in the (possibly multidimensional) torus defined by the array (or geometry) `AG`.

```
      TP = GEOMMAP(AG,UP)
```

`TASKMAP` is a scalar valued function which converts a coordinate point in a torus defined by the array (or geometry) `AG` into a user processor number.

```
      UP = TASKMAP(AG,TP)
```

In the two mapping functions, `AG` is either a defined geometry or a distributed array, `UP` is an integer value that represents the user processor (usually `MY_PE()`, but could be any value in the range 0..`N$PES`-1), and `TP` is an array that represents the processor coordinate in the topology.

These mapping functions can be used to determine which neighbors are to the north, south, east, or west. For the next example assume that north and south are defined as being governed by the first dimension of the geometry. Then

```
CDIR$ GEOMETRY G(:BLOCK,:BLOCK)
      INTEGER COORD(2)
      INTEGER NORTH, SOUTH, EAST, WEST
C Find the northern processor
      COORD = GEOMMAP(G, MY_PE())
      COORD(1) = COORD(1) - 1
      NORTH = TASKMAP(G, COORD)
```

gives a rough approximation of how to calculate a task's neighbors. Note that this code does not manage the boundary conditions that may be necessary with a toroidal or mesh topology. The toroidal wrap-around may be obtained by calculating the north/south coordinate modulo `MY_PE()`. The mesh topology would have to be handled with additional code, since some elements have no neighbor on one or more sides.

10.0 Message Passing

Many problems are naturally expressed using message passing and many codes are currently written for MPP machines which currently use message passing. This programming model supports the PVM (Parallel Virtual Machine) message passing model. It is a public domain, portable model.

PVM has been designed to be a general purpose model which can be used to communicate between heterogeneous systems as well as tightly coupled processors such as the CRI MPP while utilizing the most efficient means of transferring data. The software consists of a library of PVM functions (with Fortran and C interfaces.) The library includes functions for task creation, message passing, and synchronization.

In addition to the standard message passing primitives, two communication routines have often proven their worth in message passing MPP programs. They are the broadcast and multicast functions. Each function takes a scalar or array object and sends it to a set of tasks. Broadcast sends the object from one task to all tasks, multicast sends it from one task to a list of tasks. It seems useful to allow multicast to send an object to a neighborhood or dimension of a geometry, or to an arbitrary list of tasks. Broadcast is directly supported on the MPP, while multicast is still under consideration.

PVM is being developed by the Oak Ridge National Laboratory and the University of Tenessee, under the direction of Jack Dongarra. For more information on PVM, see [8].

11.0 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. With a small additional amount of work, very fast codes may be written.

Since real programs cannot be shown here, a smaller example is shown to demonstrate the directives in the model. A standard matrix multiply subroutine (note that there is a `MATMUL` intrinsic which should be used whenever possible; it is even faster) is first decorated with directives in a straightforward manner. The resultant code will run quite fast in parallel. These changes are entirely mechanical and require very little analysis. A further set of changes which require some knowledge of the code are added. These will further improve the code by increasing data locality. The final code segment is followed by an explanation of how it will be optimized by the compiler.

```
SUBROUTINE MATMUL(A, B, C, L, M, N)
DIMENSION A(L,M), B(M,N), C(L,N)

DO K=1,N
  DO J=1,M
    DO I=1,L
      C(I,K) = C(I,K) + A(I,J) * B(J,K)
```

```
            ENDDO
          ENDDO
        ENDDO
        END
```

As this program stands, if it were run using this programming model all PEs would execute the subroutine and no speedup would be achieved; consequently the first thing to do is to place `PARALLEL ALL` and `END PARALLEL` directives around the executable portion.

It is also useful to spread the data across the machine. This helps both with data locality and gives an large address space for big problems. Therefore the `SHARED` directive should be used. Finally the work must be shared. For this a `DOSHARED` directive must be put in.

```
      SUBROUTINE MATMUL(A, B, C, L, M, N)
      DIMENSION A(L,M), B(M,N), C(L,N)
CDIR$ SHARED A, B, C


CDIR$ PARALLEL ALL
CDIR$ DOSHARED (I,J,K) UNIFORM
      DO K=1,N
        DO J=1,M
          DO I=1,L
            C(I,K) = C(I,K) + A(I,J) * B(J,K)
          ENDDO
        ENDDO
      ENDDO
CDIR$ END PARALLEL
      END
```

Data locality is important for optimal performance. This applies to both the data distribution directives and the work distribution directives. The `(:BLOCK,:BLOCK)` distribution has been chosen here for `C`, `(:BLOCK,:)` for `A` since whole columns of `A` are used at a time, and `(:,:BLOCK)` for `B` for analagous reasons. Finally, since the data is distributed, the work should also be distributed. The `DOSHARED` directive is therefore aligned with the array `C`.

```
      SUBROUTINE MATMUL(A, B, C, L, M, N)
      DIMENSION A(L,M), B(M,N), C(L,N)
CDIR$ SHARED A(:BLOCK,:), B(:,:BLOCK), C(:BLOCK,:BLOCK)


CDIR$ PARALLEL ALL
CDIR$ DOSHARED (I,J,K) ON C(I,K)
      DO K=1,N
        DO J=1,M
          DO I=1,L
            C(I,K) = C(I,K) + A(I,J) * B(J,K)
          ENDDO
        ENDDO
```

```
      ENDDO
CDIR$ END PARALLEL
      END
```

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

Matrix multiply is a difficult problem for a distributed memory machine because it inevitably requires non-local data. The CRI MPP machine has specialized hardware which can asynchronously fetch data (blocks of contiguous data are faster but constant stride and gather operations are supported as well) from remote processors. The compiler will double buffer this example so that blocks of data are transferred to local memory while previously transferred blocks are operated upon. The example will run even faster if the `J` and `I` loops are interchanged so that all of the data transferred for each array is contiguous in memory.

Appendix A Programming Model Directives and Functions

This section summarizes the syntax of MPP 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, an array, or a named common block enclosed in slash marks ("/"). The notation α_i represents a keyword from the selection: `BLOCK` or, `BLOCK(N)`, optionally with weights, or "`:`". The expression $(\alpha_1, \alpha_2, \ldots, \alpha_r)$ may only be used with array names or geometry labels, and with array names it is optional. The notation `[RANDOM]` indicates that the keyword `RANDOM` is optional. The notation $f_i(\bar{I})$ indicates an index expression of the form $a_{1i}I_1 + a_{2i}I_2 + \ldots + a_{ni}I_n + b_i$ where a_{ji}, I_j, and b_i are all 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. Parallel Execution

```
CDIR$ PARALLEL ALL
CDIR$ END PARALLEL
```

3. Loops

```
CDIR$ DOSHARED(I1,I2,...,In)ON A(f1(Ī),f2(Ī),...,fr(Ī))
CDIR$ DOSHARED (I1,I2,...,In) [RANDOM] UNIFORM
CDIR$ DOSHARED (I1,I2,...,In) [RANDOM] CHUNKSIZE(N)
CDIR$ DOSHARED (I1,I2,...,In) [RANDOM] NUMCHUNKS(N)
CDIR$ DOSHARED (I1,I2,...,In) [RANDOM] GUIDED(N)
CDIR$ NO BARRIER
```

4. Synchronization Primitives

```
CDIR$ BARRIER
      CALL LOCKON(lock)
      CALL LOCKOFF(lock)
      L = LOCKTEST(lock)
      CALL EVPOST(event)
      CALL EVWAIT(event)
      CALL EVCLEAR(event)
      S = EVTEST(event)
CDIR$ CRITICAL
CDIR$ END CRITICAL
```

5. Input and Output

```
CDIR$ SHARED_IO
```

6. Subroutines

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

7. Intrinsic Functions

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

Parallel Prefix Functions

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

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)
```

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(AG,D)
TP     = GEOMMAP(AG,UP)
UP     = TASKMAP(AG,TP)
```

Appendix B 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. While it is not an option to implement a full Fortran 90 compiler for MPP in the time allowed, two basic elements of the Fortran 90 language will be available to MPP users. The first is array assignment statements, for example, `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 MPP. The Fortran 90 intrinsic functions which will receive earliest consideration for implementation are:

1. `ALL`
2. `ANY`
3. `COUNT`
4. `CSHIFT`
5. `DOT_PRODUCT`
6. `EOSHIFT`
7. `MATMUL`
8. `MAXLOC`
9. `MAXVAL`
10. `MERGE`
11. `MINLOC`
12. `MINVAL`
13. `PACK`
14. `PRODUCT`
15. `RESHAPE`
16. `SPREAD`
17. `SUM`
18. `TRANSPOSE`
19. `UNPACK`

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

Appendix C Future Directions

With any project there are always changes which may provide improvement. There are some additions which if designed carefully might make it easier to solve certain problems well. This section broadly outlines some features that might eventually be useful as part of the model.

C.1 Data Alignment

It is often useful to be able to align part of an array with a different part of another array. In this model as it is currently defined all arrays start on processor zero with a zero offset.

Alignments may be simple integer offsets, offsets with strides, linear functions, or specified by arbitrary functions. The future direction for this model, as it applies to `cf77` is only simple integer offsets. More may be attempted when used in conjunction with Fortran 90.

C.2 Task Teams

There appears to be a need for two different styles of task team, because of two different, but related uses. The first use is to separate functionally different segments of an application, for example, computational fluid dynamics, structural mechanics, and computational chemistry. This is a high-level heterogeneous functional decomposition.

The second potential use might be described as an intermediate level homogeneous decomposition. Problems exist that naturally decompose into parallel collections of smaller problems, but the smaller problems are still too large for execution on a single PE. Memory constraints might be cited as a concern, i.e. PE memory is too small. Or it might be that there are relatively few of these medium weight work collections, and each collection could itself be executed on a small number of PEs as if it were a task.

Task teams must provide data objects that are local to the PEs in the team, as well as being private to the team. Each team should have access to memory in its parent (grandparent, etc.) team and to shared data. No team should have access to a peer (sibling) team.

C.3 DOACROSS Loops

`DOACROSS` loops are a synchronized form of parallel loop. They have features and constraints that fall somewhere in between private and shared loops. Iterations have the ordering property of private loops, that is, each loop execution may be started only after all of the previous iterations have been started. But like shared loops one iteration need not complete before the next may be started. A distribution mechanism for `DOACROSS` loops would have to be self-scheduled and distribute in sequential order, with blocks of size one.

`DOACROSS` loops use a synchronization mechanism that is similar to events, and could in fact be implemented with them. The perspective, however, is that the event is relative to the iteration rather than some absolute event. Thus

```
CDIR$ DOACROSS (I)
```

```
          DO I = 1, N
             . . . (parallel work) . . .
CDIR$        WAIT
             . . . (dependent work) . . .
CDIR$        SEND
             . . . (parallel work) . . .
          END DO
```

would be the simplest DOACROSS loop. The WAIT directive indicates that execution may proceed up to but not beyond this point before the previous iteration has progressed beyond the corresponding SEND directive. Exactly one WAIT must be executed for each SEND directive executed in an iteration.

Since there is only one SEND and one WAIT in this example, the directives need not be numbered. If more than one protected region of dependent work is to be specified, the directives must be numbered. The numbers (or synchronization labels) must be unique, but need not occur in any particular order.

If the synchronization must take place over more than one loop iteration, a SPAN=m qualifier may be added to the WAIT directive. The symbol m represents the number of iterations the synchronization must span. For example,

```
CDIR$ DOACROSS (I)
      DO I = 1, N
CDIR$    WAIT SPAN=2
         X(I) = F(X(I-2))
CDIR$    SEND
         ...
      END DO
```

would insure that the loop execution didn't violate the dependence in the assignment statement. Conditional SEND expressions would also be permitted. Their syntax is

```
CDIR$ SEND [n] IF(condition)
```

where n, of course, is optional, and a user does not type the square brackets.

C.4 Dataflow Loop Evaluation

When a program has several self-scheduled loops in series and there are no dependences between the loops, one can simply remove the barriers at the end of all but the last loop to achieve a high efficiency. When dependencies exist between successive loops, it is undesirable for a task to stall waiting for other tasks to enter a barrier, but the barrier cannot be removed without some other form of synchronization. An iteration level dataflow mechanism would allow a task to begin the next loop before others had reached the barrier, assuming the iterations from the previous loop that caused the dependency had been completed. On the down side, such a mechanism would have to be explicitly indicated by the user because its safety often cannot be determined by static analysis. It is unclear whether the overhead of this mechanism would be low enough to justify its use.

Bibliography

1. "CF77 Compiling System, Volume 4: Parallel Processing Guide," SR-3071 4.0, Cray Research, Inc., Eagan, Minnesota, 1991.
2. "CM Fortran Reference Manual," Thinking Machines Corporation, Cambridge, Massachusetts, 1991.
3. "CRAY Y-MP, CRAY X-MP EA, and CRAY X-MP Multitasking Programmer's Manual," SR-0222 F-01, Cray Research, Inc., Eagan, Minnesota, 1991.
4. 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.
5. F. Thomson Leighton, "Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes," Morgan Kaufman Publishers, San Mateo, California, 1992.
6. "TC2000 Fortran Language Reference," BBN Advanced Computers Inc., Boston, Massachusetts, 1990.
7. 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.
8. Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam, "A User's Guide to PVM - Parallel Virtual Machine," Oak Ridge National Laboratory, ORNL/TM-11826, July 1991.
9. C. Polychronopolous and D. Kuck. "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Computers," IEEE Transactions on Computers, 36, December 1987.