# Cedar Fortran* and Other Vector and Parallel Fortran Dialects

**Mark D. Guzzi**, David A. Padua, Jay P. Hoeflinger, Duncan H. Lawrie**

**University of Illinois at Urbana-Champaign**
**Center for Supercomputing Research and Development**
**305 Talbot Laboratory / 104 S. Wright Street**
**Urbana, Illinois 61801**

ABSTRACT

The introduction of vector processors and multiprocessors punctuate the most dramatic changes in Fortran and its dialects. The emerging generation of supercomputers utilize both vector processing and multiprocessing simultaneously. The challenge is to provide language constructs and software tools that will allow the programmer to easily exploit the capabilities of the machine.

This paper will outline the development of vector and multiprocessor language constructs in Fortran. The significant architectures, their languages, and optimizers will be described. The paper concludes with a description of Cedar Fortran, the language for the Cedar Multiprocessor under development at the University of Illinois, Urbana-Champaign. Cedar is a hierarchical, shared-memory, vector multiprocessor. As such, its language, Cedar Fortran, contains many of the language features that will be described for vector processors and multiprocessors.

## 1. Introduction

The study of supercomputers and their languages is very much a study in evolution. It is a study of gradual development punctuated with an occasional dramatic change. The machines and languages that prosper are those most suited to the current environment rather than those with the most aestheticly pleasing characteristics. Through this development, Fortran has proven to be a hearty species that refuses to become extinct. As A. Perlis writes, "Fortran is not a flower, but a weed. It is hardy, occasionally blooms, and grows in every computer." It adapts to changes in the landscape, and it survives attempts to be supplanted with more attractive languages. This paper will describe vector and parallel machines and their corresponding Fortran dialects. The relationship between language features and machine architecture will be explored.

Let us examine the history and development of the Fortran family tree. Fortran was designed to be simple and efficient for executing numeric programs on a uniprocessor system. It was developed in 1958 by an IBM software team headed by John Backus [1]. This early programming language bore some resemblance to the assembly language that it was intended to replace. Possessing limited control structures and cumbersome I/O statements, Fortran programs were sometimes difficult to write and read, but Fortran compilers produced fast and efficient code. The Fortran 66 standard did little to alleviate the coarseness of the original Fortran. It was not until the Fortran 77 standard that Fortran became somewhat more palatable to the users of more modern programming languages.

The introduction of vector processors and multiprocessors punctuate the most dramatic changes in Fortran and its dialects. Vector statements have been standardized to the extent that they will be part of the next Fortran standard. No consensus has yet been reached with regard to language extensions for multiprocessing. Some proposals for parallel extensions to Fortran will be presented later in this paper. The emerging generation of supercomputers utilize both vector processing and multiprocessing simultaneously. The challenge is to provide language constructs and software tools that will allow the programmer to easily exploit the capabilities of the machine.

*This work was supported in part by the National Science Foundation under Grant No. US NSF MIP-8410110, the US Department of Energy under Grant No. US DOE DE-FG02-85ER25001, and by a donation from the IBM Corporation.

**This Author may be reached at:
Encore Computer Corporation
257 Cedar Hill Street
Marlborough, MA 01752-3004

This paper will outline the development of vector and multiprocessor language constructs in Fortran. The significant architectures, their languages, and optimizers will be described. The paper concludes with a description of Cedar Fortran [2] [3], the language for the Cedar Multiprocessor [4] under development at the University of Illinois, Urbana-Champaign. Cedar is a hierarchical, shared-memory, vector multiprocessor. As such, its language, Cedar Fortran, contains many of the language features that will be described for vector processors and multiprocessors.

## 2. Vector Processing

With the advent of vector supercomputers in the early 70's, a new branch began to form in the Fortran Family tree, a branch that would later be spliced back into the main development by the Fortran 8x standard. The vector/array processor system was the general model for most early supercomputers. In order to utilize the power of these new vector machines, statements were added to Fortran to specify vector operations. The first such machine was the Illiac IV [5] completed in 1971. Several different Fortran compilers were developed for this machine. The first was the Burroughs Illiac IV Fortran [6] which used *control vectors* as array subscripts. The elements of a control vector took on values of .true. and .false. A value of .true. indicated that the operation should be performed for the corresponding array element. A * denoted a control vector of any length with all elements set to .true. Thus, the code fragment

```
      Real A(100), B(100), C(100)
      do 10 i = 1, 100, 2
            M (i) = .true.
            M (i+1) = .false.
10    continue
      A(*) = B(*) + A(*)
      C(M(*)) = B(M(*)) + A(M(*))
```

adds corresponding elements of arrays A and B and assigns the result to array A. While only the odd elements of A and B are added and assigned to the odd elements of C.

Control vectors were somewhat cumbersome to use for complicated vector expressions. The next compiler for the Illiac IV, the IVTRAN Compiler [7] of 1973, took another approach to expressing vector operations. All assignment statements were written in standard Fortran, using array subscripts. To indicate vector operations, this language used a generic vector loop statement `do for all` to indicate that a loop should be compiled as vector instructions. IVTRAN even allowed subroutine and function calls within the body of the `do for all` which were recursively expanded in line to verify the correctness of the loop. This verification consisted of range checking and insuring that all statements could be vectorized.

Unlike the serial Fortran do loop statement, the `do for all` loop statement used an index set instead of a single index variable to specify the execution range. The index set consisted of an n-tuple of index variables and an n-dimensional range expression indicating the range of each variable in the index set. `Do for all` loops could not be nested and only limited conditional statements could be used within the body of the loop. The following code shows a simple, properly formed `do for all` statement:

```
      Real A(10,20), B(10,20)
      do 10 for all (i,j) / [1...10].C.[1...20]
            A(i,j) = B(i,j) + A(i,j)
10    continue
```

The `[1...10].C.[1...20]` specifies the range of the index variable `i`, 1 through 10, and the index variable `j`, 1 through 20. The `.C.` stands for *cross*, the cartesian cross product. Thus, a grid of values is specified

that encompasses all elements of A and B. Complementing IVTRAN, the IVTRAN Paralyzer[8] was the first source-to-source restructuring parallel optimizer. Starting with standard Fortran, the paralyzer transformed do loops and perfect loop nests into IVTRAN do for all loops. This optimizer was significant because it utilized dependence analysis to determine if loops could be safely parallelized.

Pipelined vector processors also emerged in the early 1970's: the Texas Instruments Advanced Scientific Computer (ASC) [9] in 1972 and the Control Data Corporation Star-100 [10] in 1973. The TI-ASC NX Fortran[11] Compiler was one of the first vectorizing compilers developed. This compiler could take standard Fortran 66 and produce vectorized code. The language did have a few minor vector extensions, such as triplets (see below), but the use of these extensions was not required for vector code to be generated.

Lawrence Livermore Labs developed an extented Fortran language, Vector LRLTRAN [12] for the CDC Star-100. Starting with LRLTRAN (a Fortran dialect also developed at Lawrence Livermore Labs), vector extensions were added specifically targeting the Star-100 architecture. All vectors were restricted to one-dimensional arrays. Vector assignments, vector expressions, vector functions, and control vectors were all implemented, but all vector operations required contiguous access of stride one.

Vector LRLTRAN also provided a facility for dynamically equivalencing vectors and subvectors. Given an array A(100), a subvector B of 10 elements could be dynamically defined using the statement:

```
vector (B, A((11, 20)))
```

B(1) now refers to element A(11), B(2) refers to A(12), etc. Since vectors are one dimensional and double sets of parentheses are used to specify vector ranges, no syntactic ambiguity exists between vector statements and two dimensional arrays, but the programs can be confusing to the programmer if he does not pay careful attention to the syntax.

The stride and dimensionality restrictions of Vector LRLTRAN made working with matrices cumbersome. These restrictions stem from the desire to translate vector operations in LRLTRAN directly into the vector machine instructions of the Star-100. Additionally, the use of commas to specify ranges of vectors was confusing and ensured that the current notation could never be generalized to express operations on multi-dimensional arrays as vector operations.

New vector statements, operators, and notations were introduced in 1973 by the IBM Vectran [13] and the BSP Fortran [14] compilers. Triplet expressions, identify statements, and where statements replaced control vectors and other vector notations. These vector statements are part of the proposed 8x standard. The triplet notation consists of three expressions separated by colons. These expressions indicate the range of execution of this statement and correspond to the beginning, end, and stride – as in a Fortran do loop. If the beginning of the triplet is omitted, it is assumed to be the beginning of the array; if the end is omitted, it is assumed to be the end of the array. Unless otherwise specified, the stride is assumed to be 1, and it may be omitted along with its separating colon. This notation makes it possible to assign a section of one array to a section of another array or array section expression. For example, the following instructions first assign all of A to be the values of B (a full array assignment). The next statement assigns A(1) the value 6 and A(2) the value 8 leaving the other elements unchanged:

```
integer A(4), B(4), C(8)
data B/2,4,6,8/, C/1,2,3,4,5,6,7,8/
A = B
A(1:2) = B(3:4)
```

A stride may also be used, so the following statement

```
A(1:3:2) = B(2:4:2) * C(3:8:4)
```

is equivalent to

```
A(1) = B(2) * C(3)     A(3) = B(4) * C(7)
```

Array sections can also be specified for multidimensional arrays with one triplet for each dimension of the array, and triplet notation may be mixed with the normal array index notation. The only restriction is that any two sections appearing together in the same statement must be conformable, ie. both sections must have an equal number of elements in each corresponding dimension. For example:

```
integer A(4,4,4), B(4,8), C(4,6,4,4)
A(1:3:2,2:4:2,1)=B(1:2,2:4:2)*C(3:4,5,2:4:2,1)
```

All the array sections in the above example are 2 x 2 sections. Expanded, they are:

```
A(1,2,1) = B(1,2) * C(3,5,2,1)
A(3,2,1) = B(2,2) * C(4,5,2,1)
A(1,4,1) = B(1,4) * C(3,5,4,1)
A(3,4,1) = B(2,4) * C(4,5,4,1)
```

Triplets are not sufficient to express all vector operations. Specifically, conditional operations that were possible with control vectors cannot be performed with triplets. The where statement, another Vectran construct, is a conditional vector assignment that allows more flexibility than control vectors. The syntax of the where statement is shown in Figure 1. The where statement first evaluates a logical array expression. The statements in the body of the where are executed for each index value for which the logical array expression evaluated .true. The body of the where statement, either a single statement or a block, contains only array assignments. The right hand side of every array expression in the body must be conformable to the logical array expression. This is a vector statement, so the logical array expression and all right hand side (rhs) expressions in the body are evaluated before the assignments are performed. If the otherwise block is present, the array assignment statements of the otherwise block will be performed for every corresponding element of the logical array expression of the where statement whose value is .false.

```
where (<logical_array_expr>)
        <array_assignment_statements>
[otherwise
        <array_assignment_statements>]
end where
```

Figure 1. Where Statement Specification

A simple example of the use of the where statement would be to zero all array elements that are negative. In this example only the elements of a that have values less than zero will be set to zero. All the other elements will remain unchanged.

```
integer A (100)
where (A(1:100) .LT. 0)
      A(1:100) = 0
end where
```

The combination of triplets and where statements is still not adequate to express all vector operations. Particularly, it is not possible to express operations that have a regular stride in physical memory but an irregular stride in array indices. Assigning to the diagonal elements of a two-dimensional array is a simple example. Given an array, a(10,10); it is not possible to assign just the diagonal elements using triplets and where statements without control vectors even though the distance between all the referenced elements is the same (stride = 11 in this case). The identify statement of Vectran and Fortran 8x [15] allows aliasing to a part of an array so that such operations can be performed. To access the diagonal of array A, the following code would be used:

```
real A(10,10)
identify (Diag(I) = A(I,I), I = 1, 10)
Diag(:) = 1
```

Although sufficient, the identify statement introduces aliasing into the program. The forall statement, which has been removed from the proposed 8x standard, provides the same functionality as identify without the additional aliasing. The forall looks very much like an identify statement, except that it performs operations within the statement instead of just aliasing. The equivalent code using the forall statement is:

```
real A(10,10)
forall (I = 1:10) A(I,I) = 1
```

The forall also allows conditional assignment making it possible to express complicated vector operations with a single statement. In the following example, the diagonal elements of a matrix are tested, and any element that is negative is set to zero:

```
integer a(10,10) i
forall (i=1:10, a(i,i) .LT. 0) a(i,i) = 0
```

The forall syntax is shown in Figure 2 and is essentially the same as the the IVTRAN do for all statement.

```
forall ([<loop_spec>]+...[,<log_array_exp>])
   <array_section_expr> = <array_section_expr>
where
<loop_spec> -> <integer_variable> = <triplet>
<log_array_exp>->same as in Where statement.
```

Figure 2. `Forall` Statement Specification

Most compilers for vector machines from the late 1970's and early 1980's accept standard Fortran as input, and vectorization is done by the compiler. The Cray CFT compiler [16] (the Cray 1 Fortran compiler) accepts standard Fortran and directives from the programmer. The compilers for the Cyber 205 (Cyber 205 Fortran [17]) and the Fujitsu VP-200 (Fujitsu Fortran 77 [18]) are also full vectorizing compilers, but the Cyber 205 Fortran also supports the Vectran-style vector extensions. Recently, many vectorizing compilers have been developed for various architectures: IBM VS Fortran [19], Alliant FX/8 Fortran [20], NEC SX Fortran [21], and others.

Also during the 1970's, software tools that perform program restructuring for vectorization and then report back to the user were introduced; tools such as Parafrase [22] [23] [24], developed at the University of Illinois, and PFC [25], developed later at Rice University. The goals of such tools were to provide very powerful system-independent vectorizers, to evaluate the effectiveness of new optimizations, and to involve the user in the optimization process with the ultimate goal to educate the programmer to enable him to write better programs.

## 3. Multiprocessing

The introduction of multiprocessor systems has also had a dramatic affect on Fortran dialects, but no standardization of multiprocessor constructs has been attained. Multiprocessor systems offer greater flexibility than vector processors, but they also offer a greater challenge in programming and compiling. Unlike compiling and programming for vector machines, which involve localized decisions, compiling and programming for multiprocessors often involve global consideration of the program. Additionally, utilization of a multiprocessor often requires consideration of both the operating system environment and the machine architecture making standardization difficult.

From the large number of multiprocessor systems that have been designed or proposed, two basic multitasking methods have emerged to increase the performance of a single program: macrotasking and microtasking. Macrotasking involves breaking the problem into large chunks, called *tasks* that can execute more or less independently on the multiple processors. Macrotasking is usually realized with the familiar `fork/join` constructs of [26]. The `fork` construct creates a new task that will be scheduled for execution by the operating system. Creating a new task is generally a very expensive operation. The useful work performed by the task must be sufficiently large to compensate for this startup overhead if macrotasking is to be efficient.

A very interesting system was the Denelcor HEP [27] developed in the late 1970s. A single HEP cpu executed multiple instruction streams in parallel. The cpu was time-multiplexed between the instructions streams. Multiple pipelines and register sets allowed the HEP to switch rapidly between processes without saving process state information. A minimum of eight processes was required to keep the cpu fully utilized. Therefore, even on a uniprocessor system, the HEP relied on macrotasking to increase performance.

The HEP Fortran contained extensions to utilize the multitasking capabilities of the machine. Two new statements, `create` and `resume`, were added to the Fortran 77 base language. The `create` statement allowed the parent process to invoke a subroutine as a new process; the `resume` statement allowed a called routine to force its parent to continue while the called routine executed in parallel. The following two code segments in HEP Fortran would both result in subroutine `subr` executing in parallel with its parent process.

```
create subr (x,y,x)          call subr (x,y,z)

subroutine subr (x,y,z)      subroutine subr (x,y,z)
<some computation>           resume
return                       <some computation>
                             return
```

Many current supercomputers consist of a relatively small number of very high performance vector processors. They rely on multiprocessing to provide higher performance than single-processor vector systems. Both the Cray-XM/P [28] and the ETA-10 [29] provide macrotasking library routines in their Fortran languages. These systems require the programmer to divide the work of his program into tasks that may execute in parallel. No automatic task generation is done by the compiler. The library routines instantiate new tasks at the subroutine level. For example, the following piece of Cray-XM/P Fortran code results in two tasks executing the same subroutine with different parameters:

```
external subr
integer task1(3), data(1000)
call tskstart (task1, subr, data, 1, 500)
call subr (data, 501, 1000)
```

The first call creates a new task executing `subr` with `data` elements from 1 to 500. The second call invokes `subr` directly with `data` elements from 501 to 1000. The `task1` array is a control array used by the macrotasking library.

Along with the ability to create multiple instruction streams, effective multitasking also requires some method of controlling (synchronizing) the executing tasks. For the purpose of this paper we distinguish between *critical section* synchronization and *event-waiting* synchronization. A critical section implies that entrance to a particular section of code must be restricted so that only one processor may execute it at a time. This is done to prevent processors from interfering with one another. *Event-waiting* synchronization imposes an ordering upon asynchronous events. The *producer-consumer* problem is a good example of event-waiting synchronization. Critical section synchronization is usually handled with a semaphore [30] mechanism. Event waiting synchronization may be handled in a variety of ways: a processor may wait until the value of a particular variable becomes zero or explicit event posting and waiting mechanisms may be used.

The HEP provided synchronization using *asynchronous variables*. An asynchronous variable was any variable whose name began with a dollar sign ($). These variables had a `full/empty` synchronization bit associated with them. An asynchronous variable could only be read if its synchronization bit was `full`; it could only be written if its bit was `empty`. A read automatically set the bit to `empty`, and a write automatically set the bit to `full`. An instruction stream wanting to read or write a such variable would wait until the synchronization bit was in the proper state. A section of code bounded by a read and a write of an asynchronous variable therefore became a critical section. These variables could also be used for event-waiting synchronization.

Synchronization in Cray-XM/P Fortran is done with calls to the runtime system. Routines are provided for locking (critical section) and event-waiting synchronization. Figure 3 compares the HEP and the Cray-XM/P synchronization methods for a simple producer-consumer synchronization problem and a simple critical section problem.

Another consideration in macrotasking is the nature of the program data. Multiple tasks executing a single program require that multiple data spaces also exist. Cray Fortran uses stacks to allocate data local to subroutines and functions. This departs from the more common static allocation of local data in typical Fortran implementations, but this change is necessary to accommodate macrotasking. In both Cray and HEP, a local variable allocated on the stack can be passed by reference to newly spawned tasks. Care has to be taken, however, that variables are not deallocated from the stack before all tasks finish using those variables.

`Common` blocks also require some consideration. `Common` blocks are used in Fortran to share data across subroutine calls. In a macrotasking environment, is there one `common` block for the entire program or one per task? In the HEP only one `common` block exists per program and is shared by all tasks. In the Cray two types of common are allowed. Plain `common` blocks are shared by all tasks in the program. `Task common` blocks are allocated one copy per task. The variables within a `task common` could be passed by reference to newly spawned tasks.

Microtasking exploits parallelism at a more local level than macrotasking. Microtasking is often used to execute iterations of a loop in parallel. In addition to loop parallelism, speedups can be achieved by overlapping segments of sequential code. This operation is called low-level spreading if the spreading is done on a statement-by-statement basis; it is called high-level spreading if the spreading is performed upon large instruc-

| | HEP | Cray-XM/P |
|---|---|---|
| Producer-Consumer | | |
| Initialization | `integer $emp`<br>`integer $fll`<br>`purge $emp,$fll` | `integer emp, fll`<br>`call evasgn (emp)`<br>`call evasgn(fll)`<br>`call evpost(emp)` |
| Producer | `e = $emp`<br><produce><br>`$fll = 1` | `call evwait(emp)`<br><produce><br>`call evpost(fll)` |
| Consumer | `f = $fll`<br><consume><br>`$emp = 1` | `call evwait(fll)`<br><consume><br>`call evpost(emp)` |
| Critical Section | | |
| Initialization | `integer $lkvar`<br>`purge $lkvar` | `integer lkvar`<br>`call lockasgn(lkvar)` |
| Section | `l = $lkvar`<br><Crit. Sect.><br>`$lkvar = 1` | `call lockon(lkvar)`<br><Crit. Sect.><br>`call lockoff(lkvar)` |

Figure 3. HEP vs. Cray Synchronization

tion streams [31]. This optimization adds to the overall speedup of the program because it overlaps instructions that are not vectorizable. Microtasking is efficient on such small execution units because it does not incur the large task creation overhead of macrotasking. Instead of creating new tasks for each new parallel execution stream, a fixed number of computational resources, either "helping" tasks or real processors, are allocated at the beginning of the program. These resources remain idle until parallel code is encountered by an active task. At this point, the "helping" resources join in the parallel execution. Once a resource joins in the execution, it is committed until the parallel execution completes. The overhead for starting this parallel execution is much lower than in macrotasking.

In 1979, Burroughs proposed the FMP multiprocessor and a companion extended Fortran (called FMP Fortran) [32]. The FMP was never built, but its design foreshadowed the development of future multiprocessor systems. The machine was comprised of 512 processing elements with local memory. All processors could also access a much slower shared extended memory. Additionally, a processor could broadcast to the other processors for quick distribution of information. The early strategy of FMP Fortran was to allocate all variables in the local memory of the processor unless instructed to do otherwise. FMP Fortran introduced a new concurrent construct called the `doall`, one of the first uses of a microtasking construct. The `doall` allowed the specification of multiple index variables or n-tuples of indexes, called *domains*. All iterations of the `doall` were considered discrete and independent. The FMP `doall` format is shown in Figure 4. This `doall` required the user to specify the intended use of variables. The `using` clause identified input variables, and the `giving` clause identified output variables.

Any variable that does not appear in the `using` or `giving` clause is allocated locally to each processor. The `using` and `giving` clauses in the FMP `doall` statement force the user to provide the FMP Fortran compiler the dependence information necessary to make proper allocations, and thus relieves the compiler of this task. Another important feature of this `doall` is that no information is passed between the iterations of the loop. At the beginning of the loop, the state of the extended memory is "frozen." All modifications are local until the end of the `doall`, then the variables listed in the `giving` clause are written back. This value/restore form of execution eliminates the possibility of asynchronous side-effects (loop executions are determinant), but it also limits the types of loops that can be converted into a `doall`. Later versions of the FMP `doall` allowed data sharing and asynchronous operation.

```
doall <domain>; using <variable list>
        <loop body>
enddo [<domain>]; giving <variable list>
```

Figure 4. FMP `Doall` Specification

Microtasking has also been used in a number of current multiprocessor systems. In some of these systems, the processors execute in a more independent fashion than in the FMP. In all of these systems, data may be shared between the iterations of the loop. With data sharing, nondeterminacy may be introduced into the execution of the loop. In loops where the iterations are completely independent (or require some critical section synchronization), a more general `doall` loop statement may be used. The iterations are executed by multiple processors and no guarantee is made about the order of execution of the iterations of the loop.

Many loops, however, have some serial component. The natural ordering of the loop must be preserved for the loop to execute correctly. The `doacross` loop is a parallel loop that assumes synchronization from left to right [33] [34] [35], meaning that dependences may exist between iteration `I` and some previous iteration(s). The iterations of the loop are scheduled "horizontally" – meaning that no processor will begin execution of iteration `I` until iteration `I-1` has been started. (Horizontal scheduling is also called cascading execution.) Additionally, an iteration may have to wait during its execution for a previous iteration in order to satisfy dependences. In other words, a producer-consumer relationship exists between the iterations of the `doacross` loop.

Among the microtasking libraries of current multiprocessor systems, a great variety of routines exists with varying levels of sophistication. The Cray-XM/P supports a `doall` style loop in its microtasking system as well as high-level spreading. Sequent Fortran [36] provides a `doacross` style loop and microtasking `fork/join` style constructs. The microtasking constructs of both are supported in software. The Alliant FX/8 multiprocessor [37] and the IBM RP3 multiprocessor [38] support microtasking `doacross` loops at the hardware level. Special hardware that schedules the iterations of concurrent loops is incorporated into their architectures. In some systems (Cray, Sequent), the user must insert statements or compiler directives to use the microtasking. In the Alliant, the compiler automatically converts serial `do` loops into microtasking loops. One of the major advantages of loop parallelism and high level spreading is that the block-structure of the program is maintained while parallelism is exploited. Macrotasking usually requires a complete rewriting of the program to break it into asynchronous tasks.

It may initially seem that microtasking is superior to macrotasking for exploiting concurrency, but each method has advantages and disadvantages. In the development of multiprocessor systems, both microtasking and macrotasking have been used, and they both continue to be used. The appropriate method of execution depends both on the underlying architecture and the nature of the application.

Microtasking has the advantage of reducing overhead and exploiting parallelism at a finer grain. Microtasking also has two major disadvantages. First, the number of helping resources is usually fixed at compile time or at program startup time (this is done to reduce runtime overhead). Thus, the system does not adapt well to data size or system load. Second, the combination of resource commitment with the non-determinacy of the number of helping tasks that will actually participate in the execution of the parallel code restricts the synchronization that may be used safely. Non-cascading synchronization in microtasking may be disastrous because a task performing a waiting synchronization may be the only task in the execution. Since the resource is committed to its instruction stream, it cannot switch execution streams to satisfy the wait condition. Deadlock is the result. Both of these problems can be remedied with operating system intervention, but then much of the efficiency advantage is lost.

Macrotasking, on the other hand, suffers from higher overhead and thus necessitates a coarser granularity of tasks, but it does not have the above problems of microtasking. Since macrotasking relies on operating system task management, dynamic adaptation to system load and problem size can be incorporated into the runtime system. Additionally, macrotasking does not have the deadlock problem of microtasking. When synchronization is required, a single resource can perform the synchronization and then context switch to another instruction stream. At some later time, the wait condition may be satisfied by the execution of another task, thus releasing the original task.

Many multiprocessor systems, such as the Encore Multimax[39], the Sequent Balance, and IBM Parallel Fortran 3090 [40], now have or are developing Fortran systems that incorporate both macrotasking and microtasking. Table 1 compares the capabilities of many multiprocessor systems.

This great variety of macrotasking/microtasking libraries/routines has caused some cries of despair among the user community: "Parallel pro-

grams are not portable!", "Parallel programs are more difficult to write!", etc. In response to these cries. two approaches have been taken. The first approach is to provide an automatic parallelizer – an optimizer that takes a standard sequential program and converts it into a parallel program. This is the approach taken with IBM VS Fortran and Alliant Fortran.

The other approach is to provide a generic parallel environment in which to produce parallel code. Software systems such as Linda [41] have been created which present a uniform parallel programming language that can be implemented on a variety of multiprocessor and distributed systems. In Linda, shared memory is represented as a tuple space. A tuple consists of a variable and one (scalar) or more (array) values. Variables are not written and read as in most languages. Instead, a tuple acts like a mailbox. Values are output to (received by) the tuple, read from the tuple, or removed (input) from the tuple. A value cannot be overwritten; it must be consumed as input before a new value can be output. Data items are not referenced by address, but by name. Accessing a shared variable is similar to searching a distributed database. This shared data feature coupled with elegant methods for handling multitasking make Linda a pleasant programming environment.

## 4. The Cedar Multiprocessor and Cedar Fortran

The Cedar multiprocessor is a hierarchical shared-memory supercomputer under development at the University of Illinois, Urbana-Champaign. The processors are grouped into clusters that share access to a cluster memory. Processors in one cluster may not access the cluster memory of another cluster. All processors in the machine share access to a large global memory through an interconnection network. The Cedar-1 machine comprises four clusters of eight processors for a total of 32 processors, but this hierarchical architecture may be extended to hundreds of processors. The clusters in Cedar-1 are Alliant FX/8 multiprocessor systems. Each processor is capable of performing vector operations, and the Alliant concurrency hardware allows the processors in each cluster to share the execution of concurrent loops.

The Cedar Fortran Language is based on Fortran 77 [42] and has remained mostly consistent with that standard, but many extensions have been added to provide for optimized concurrent execution. In trying to adapt Fortran to a multiprocessor, multitasking system such as Cedar, many serious problems arose because of the simplicity of the language. The Cedar extensions mostly take the form of new statements and "intrinsic" functions: vector statements, statements for expressing concurrent loops, statements and functions for synchronization, and functions for multitasking. The concurrent loop extensions mentioned above necessitated another extension to standard Fortran, block structured data scoping. Data allocation and scoping presented the greatest problem in adapting Fortran to the multiprocessor/multitasking environment. The data scoping issues are more easily understood with knowledge of the parallel control structures that have been implemented. For this reason, discussion of data scoping will be postponed until after the introduction of the control structures.

### 4.1 Array/vector Extensions

The vector extensions of Cedar Fortran provide powerful constructs for manipulating data in a vector fashion. Cedar Fortran implements three vector statement types, triplet notation assignments, `forall` statements, and `where` statements. The triplet notation and `where` statements are compatible with the proposed Fortran 8x standard. Their derivation from Vectran was described in a previous section. The `forall` statement has been included in Cedar Fortran because it provides the same functionality as the 8x `identify` statement without introducing more aliasing. The `identify` statement has been omitted in favor of the `forall` statement.

Conceptually, the triplet is a vector assignment statement, the `forall` is a vector loop statement, and the `where` statement is a vector `if` statement. These vector statements are essential to the Cedar Fortran optimizer because they allow all vectorizable instructions to be represented. Once statements have been expressed in vector form, it is much easier for the back end compiler to generate efficient code.

### 4.2 Concurrent Loops

Vector operations provide one avenue for increasing program performance, but not all code is vectorizable. Concurrent execution with the use of multiple processing resources provides another avenue of optimization. Just as vector operations can be substituted for many Fortran `do` loops, concurrent loops can also be substituted for `do` loops. Additionally, high-level spreading makes it possible to parallelize code that is not in `do` loops. Concurrent constructs can easily be integrated with vector statements to provide multiple levels of parallelism.

The language constructs used to define concurrent execution in Cedar Fortran are derived from the `doall` and `doacross` statements described in the previous section. Each of these loop types is further divided into three groups: cluster statements (`cdoall` and `cdoacross`), spread statements (`sdoall` and `sdoacross`), and cross cluster statements (`xdoall` and `xdoacross`). The distinctions between the groups will be explained below.

The syntax of the `doacross` loop in Cedar Fortran (subsequently to be referred to as CF) is similar tc the standard Fortran do loop; its form is shown in Figure 5. Provision has been made for statements that are to be executed once for each processor participating in the loop using the optional `loop` statement. Those statements that appear before the `loop` will be executed once per processor. Those statements that appear after the `loop` will be executed on every iteration. If the `loop` statement is not present, all statements will be executed once per iteration. Type statements may appear immediately after the `doacross` statement. The function of these type statements is to declare `doacross`-internal variables and arrays which may be referenced only inside that `doacross` loop. Each iteration of the loop allocates a local, private copy of these internal variables and arrays. Each variable and array that is to be internal to a `doacross` loop must be declared explicitly. Any previous declarations are superceded for the duration of the loop.

```
        doacross  [label[,]] i = e1, e2 [, e3]
            [<type_statements>]
            [<statements>]
          [loop]
            [<statements>]
   {label   <next_statement>}  |  {end doacross}
```

where

$i$ is the name of an integer variable, called the `doacross`-variable.
$e_1$, $e_2$, and $e_3$ are each integer expressions.

Figure 5. `Doacross` Loop Specification

If data or control dependences exist between the iterations of the loop, the programmer must insert proper synchronization instructions to insure the correct execution of the loop. Two routines, `advance` and `await`, are provided for `doacross` synchronization. An iteration of the `doacross` uses `await` to wait for a previous iteration. An `advance` releases the `await` of a future iteration. (These routines are very efficient because they interface directly to the Alliant concurrency bus.) If no data dependences exist between the iterations of the loop, then the programmer should use the `doall` statement instead of `doacross`. The `doall` loop is very similar to the `doacross` loop; The syntax is identical except for replacing the "`doacross`" with "`doall`". While the iterations of a `doacross` are scheduled horizontally, no order is implied in the execution of the iterations of a `doall`. This distinction is important because CF will not override the programmer's declaration of independent parallel execution. The CF compiler may, however, take advantage of this declaration and restructure the loop to achieve greater speedup. If a specific execution order is necessary for the loop to execute correctly, a `doacross` should be used.

It is sometimes desirable to exit a loop before all the iterations of the loop have been completed. A `goto` statement cannot be used to exit from a parallel loop because a `goto` would only affect a single processor. The result would be that one processor would leave the concurrent loop while the other processors continued to execute within the loop. Instead, the following two statements, `quit` and `qquit` have been provided.

```
quit [label]
qquit [label]
```

The `quit` statement causes the parallel loop to be terminated cleanly. It waits for all iterations with a loop index less than the current loop index to finish before it actually quits the loop. Without the `label`, execution resumes at the first statement after the end of the innermost loop containing the `quit`. With the `label`, execution resumes at that label, terminating all loops out to that level.

If multiple iterations of a loop perform a `quit` operation, then the iteration with the smallest iteration number (if the stride is positive) will

control the exit of the loop regardless of which iteration actually issued the quit first in real time. The qquit statement is similar to the quit statement, except that it terminates the loop immediately. (It does not wait for all previous iterations to finish.) It is a "quick quit." Caution must be taken when using qquit because previous iterations may be only partially complete when the loop terminates.

4.3 Macrotasking

In addition to the microtasking of concurrent loops, CF provides macrotasking via the macrotasking library (similar to the Cray macrotasking routines). The macrotasking routines are the interface between the CF language and the Xylem operating system [43]. This library provides routines for creating and controlling "tasks." (An entity of concurrent execution in Xylem is called a "cluster task" or just a "task" for convenience.) A task begins execution at the specified routine and continues independently until the end of the subroutine is reached. When the return statement is reached, the task is terminated rather than returning to the parent task. The macrotasking library provides routines for the creation and monitoring of these user defined tasks. These routines are described in Figure 6.

4.4 Cluster, Spread, and Cross Cluster Loops

The format of the ctskstart call may initially appear strange to the reader because it specifies a number of processors to be allocated to the new task. This number of processors is significant in the execution of concurrent loops. As was mentioned earlier, there are three forms for each concurrent loop: cdoall, sdoall, and xdoall, and cdoacross, sdoacross, and xdoacross. The "c" loops are cluster or "confined" loops, meaning that the processing resources used for the execution of the loop are confined to processors allocated to the current cluster task. For example, a cluster task with five processors is started with:

```
taskid = ctskstart (5, subr, x, y, z)
```

This creates a task for subroutine subr with arguments x, y, and z. Within subr, a program segment

```
cdoall i=1, 100
      <statements>
end cdoall
```

executes a concurrent loop using only the five processors allocated to the task at the task's creation. The execution of the loop is confined to the processor resources of the cluster task.

Also within subr, a very similar program segment

```
sdoall i=1, 100
      <statements>
end sdoall
```

has a very different meaning. In this loop, the execution is being spread to processing resources outside the current cluster task. A concurrent loop statement that begins with an "s" indicates that this is a "spread" loop. In the loop above, each iteration of the loop may be assigned to a different cluster task. If the CF user specifies his own concurrent loops, then he is responsible for proper declaration and allocation of variables. However, the compiler will provide warnings of improper user declarations. Any concurrent loops generated by the CF optimizer will have the variable declaration done automatically to match those loops.

In order to make effective use of the computational resources on Cedar, a sdoall loop should have a nested cdoall loop to bring all the processors within each task into execution. It may initially seem strange that the sdoall allocates whole cluster tasks to the iterations of the loop instead of just processors, but one must remember the hierarchical structure of Cedar. In order to reach the processors within a cluster, one must go through the cluster control hardware (Alliant concurrency bus). The sdoall/cdoall loop combination provides the programmer maximum flexibility in controlling the parallel execution of loops.

If the programmer desires to run one loop across all processors of the machine and s/he does not want to deal with the precise details of loop blocking and processor scheduling, the xdoall or xdoacross statements may be used. These "cross cluster" parallel loops take the confined processors of the given cluster task and combine them with the processors of other tasks to create a single loop with processors from different clusters taking iterations of the loop. The x concurrent loops are functionally equivalent to a s loop with an inner nested c loop, except the loop blocking and processor utilization is managed by the compiler instead of the programmer. (These x concurrent loops are currently unimplemented in CF.)

4.5 Synchronization

With the addition of concurrent loops and macrotasking, a mechanism for controlling the execution streams is required. New statements and intrinsic functions were added to CF to provide for varying granularities of synchronization. Fine grain synchronization is used between processors within concurrent loops, larger grain synchronization is used between tasks.

Cedar Fortran provides mechanisms for both critical section and producer-consumer synchronization, adopting the full set of Cray lock and event synchronization routines. These routines are classified as medium grain synchronization, appropriate for synchronization between cluster tasks, but too expensive for general use between the processors in a single task.

---

- `int = ctskstart (num_proc, sub [,arg]...)`

  Ctskstart spawns a new task, and returns an integer that identifies it. The first argument, num_proc, specifies the number of processors to be allocated to the new task for execution of any cdoall or cdoacross loops. If the value is zero, then the system is free to allocate any number of processors to the cluster task.

  The second argument, sub, is the subroutine or entry point at which execution will begin in the new task. The remaining arguments are those passed to the subroutine. These arguments may be of any type (scalar or array), but they should match the formal parameters of sub. Expressions are allowed as arguments and enclosing an argument in parentheses indicates that the argument is to be passed by value. The value returned by ctskstart will either be a positive integer corresponding to the id number of the newly spawned task, or -1 if ctskstart fails.

- `logical = ctskdone (int)`

  Ctskdone reports the state of a task created by ctskstart. The value of the argument identifies a task and must have been obtained from ctskstart. Returns .false. if the specified task has been created and has not terminated. Returns .true. otherwise.

- `call ctskwait (int)`

  Ctskwait suspends the task referencing it until the task whose identification is the argument completes execution. The value of the argument must have been obtained from ctskstart.

Figure 6. Macrotasking Library Routines

---

```
with <resource> when|if <expression>
      <statements>
end with
```

Figure 7. The With Statement

---

Besides these synchronization primitives, we find the with statement described by Hoare [44] to be useful in expressing critical section synchronization. The general form of the with block is shown in Figure 7. This construct means that the <statements> are in a critical section for the <resource>. In CF the <resource> is a variable or array element declared as sync. In the with/when form, these <statements> will not be executed until exclusive access can be granted to the <resource> and the <expression> is true. (Note that the with statement does not hold the <resource> while it is waiting for the <expression> to become true.) In the with/if form, exclusive access to the <resource> is granted and the <expression> is evaluated; if the <expression> is true, the <statements> are executed, otherwise execution continues immediately after the end with. Exclusive access is always released at the termination of each with statement.

The sync type statement declares a variable or array element, x, to have a data field, a tag field, and a hidden lock field. When referring to the data elements, one simply uses the variable name, x. The tag field is referenced as x.tag. Figure 8 shows a do loop with a subscripted-subscript transformed to run in parallel. This requires both event-waiting and critical-section synchronization. Using b(i) as the index variable causes the order of storing the array a to be unpredictable. The tag field is used to maintain the order of the accesses to the elements of a. More examples of the use of this synchronization method can be found in [45].

Although any statements may be contained in the body of a with statement, the Cedar hardware provides synchronization primitives that can be used to implement some with statements as single instructions. The with statement in the above loop, for example, can be implemented as a

single instruction in Cedar. If the operations in the body of the with are limited to only one operation each on tag and data, then the entire body of the with may be executed as a single instruction. If the body contains more operations, then additional software synchronization must be used.

The loop:

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

The equivalent parallel versions using the with statement:

```
doall i = 1, N
      with a(b(i)) if (a(b(i)).tag < i)
            a(b(i)) = c(i) + 1
            a(b(i)).tag = i
      end with
end doall
```

Figure 8. With Statement Example

4.6 Data Declarations, Scoping, and Nested Concurrent Loops

The Cedar architecture physically distinguishes two types of memory: global memory which is accessible to all processors and cluster memory which is only accessible to the processors in that cluster. Specification of location attributes for data in CF are supported by extending the standard Fortran declaration statements with two new attribute statements, global and cluster. While the architectural classifications of global and cluster are very distinct, Cedar Fortran, with the help of the Xylem virtual memory system, presents a more homogeneous virtual address space. All data that is declared locally in a subprogram and passed outside that subprogram is heap-allocated and sharable with other tasks. The cluster and global attributes define only initial locations of data. Data will be demand-paged to the physical location necessary to meet the access demands of the program. Data that is read and written by multiple tasks must be moved to the global memory. Read-only data, program code, and unshared data may move to cluster memory. The programmer must be somewhat aware of the underlying architecture to avoid excessive paging activity. Data to be shared by several cluster tasks or that is shared among the iterations of an sdoall loop should be declared global. Library routines are provided to the programmer for changing location attributes of data.

Common blocks are used in Fortran to share data across subroutine calls. In CF, there are two types of common, plain common and process common. In plain common, one copy of the common is (potentially) created for each task within the program. In process common, one copy of the common is shared by all tasks within the program. Unlike the task common of Cray Fortran, plain common of CF cannot be passed by reference to newly spawned tasks. Unlike local data which can be shared, the data within a plain common is strictly private to the task. This undesirable restriction is forced by the back-end compiler. In future implementations, it is hoped that this restriction will be lifted so that the programmer is presented with uniform data behavior where data sharing is completely dictated by visibility and scoping.

Common names may appear in the CF memory attribute type statements. This means that the common defined by this name and all of the identifiers located within it are of the indicated type. Any plain common that is not specifically given attributes will be given the cluster attribute. Any process common that is not specifically given attributes will be given the global attribute. All identifiers in a common take on the initial location attribute of the common block. The actual location may be changed by demand paging or by library routine calls.

The open scoping of identifiers in Fortran proved to be unmanageable for concurrent loops. A C-like block-structured scoping system was added to CF in order to isolate variables in a single instruction stream. Specifically, variables and arrays may be declared inside any concurrent construct. The newly declared variables will be allocated one copy per iteration of the concurrent construct, and these declarations will supersede any variable of the same name declared outside the construct. These declarations remain in effect until the construct terminates or until new declarations of a more nested construct supersede the older declarations. When a construct terminates, the declarations as they existed before the construct began are restored.

The scoping levels have similar implications for goto statements. CF allows gotos to be performed only to statements at the same scope. This essentially means that CF will not allow gotos to be performed into or out of doall or doacross constructs. Of course, this rule applies to any type of goto, either explicit, or by result of an arithmetic if or any other Fortran branching construct. Recall that quit or qquit must be used to terminate a parallel loop before all iterations have been completed.

Nested concurrent loops should allow greater exploitation of parallelism within a program. Multi-level loop parallelism can be supported under the current implementation of CF using a combination of sdoall and cdoall or cdoacross loops. The Alliant hardware controlling the cluster loop execution permits only the outermost cluster loop to execute concurrently; all inner cluster concurrent loops will execute sequentially.

When using an sdoall loop, all input and output data from the loop must be accessible outside the initiating cluster task. All such data should therefore be declared global to avoid excessive data movement (and that data cannot appear in plain common blocks). Below is a properly constructed nesting of spread and cluster loops.

```
dimension a(10,20)
global a, i
sdoall i = 1, 10
      integer j
loop
      cdoall j = 1, 20
            a(i,j) = i+j
      end cdoall
end sdoall
```

## 5. Conclusion

In this paper, we have traced the development of vector extensions to Fortran from their diverse beginnings to the current general consensus. The continuing variety of multiprocessing extensions has also been presented. Most current multiprocessor extensions are still deeply intertwined with their corresponding architectures and operating systems. Cedar Fortran is no exception. Our primary goals in designing and implementing the language were:

(1) to provide a language that easily allowed direct access to all the power of the Cedar system, both for the programmer and for source-to-source optimization.

(2) to provide a Fortran dialect in which users could easily adapt their existing Fortran applications.

In these two goals, we feel we have been reasonably successful. Our secondary goals were:

(1) to adapt the language to the needs and desires of programmers as they gain experience with parallel programming.

(2) to add to the overall knowledge and experience of parallel programming and parallel languages.

We are confident that time will provide a general consensus on multiprocessor Fortran constructs, just as it has with vector constructs. The diversity of multiprocessors is much greater than that of vector processors, and the problem is compounded by the interrelated issues of concurrent execution and proper data allocation. The convergence of ideas and attitudes may, therefore, take more time and effort.

## REFERENCES


[1] J. Backus, "The History of FORTRAN I, II, and III," *Annual History of Computers*, vol. 1,1, July, 1979, pp. 21-37.

[2] M. Guzzi, "Cedar Fortran Programmer's Manual," CSRD doc. no. 601, U of I Center for Supercomputing R and D, Jan. 1987.

[3] D. Padua and D. Lawrie, "Proposed Cedar Fortran Extensions," CSRD doc. no. 509, U of I Center for Supercomputing R and D, Sept. 1985.

[4] D. Kuck, E. Davidson, D. Lawrie, and A. Sameh, "Parallel Supercomputing Today and the Cedar Approach," *Science*, Vol. 231 (28 Feb. 1986), pp. 967-974.

[5] G. Barnes, R. Brown, M.Kato, D. Kuck, D. Slotnick, and R. Stokes, "The Illiac IV Comuputer," *IEEE Transactions on Computers*, Vol C-17 No. 8 (Aug. 1968), pp. 746-757.

[6] *Array Processing System Fortran IV Reference Manual*, Defense, Space, and Special Systems Group, Burroughs Corporation, Paoli, Penn., Change No.3, August 31, 1971.

[7] R. Millstein and C. Muntz, "The Illiac IV Fortran Compiler," *ACM Sigplan Notices*, Vol. 10 No. 3 (March 1975), pp. 1-8.

[8] D. Presberg and N. Johnson, "The Paralyzer: IVTRAN's Parallelism Analyzer and Synthesizer", *ACM Sigplan Notices*, Vol. 10 No. 3 (March 1975), pp. 9-16.

[9] L. Higbie, "Supercomputer Architectures," *IEEE Computer*, Vol. 6 (Dec 1973), pp. 48-58.
[10] *Control Data Star-100 Computer System – Hardware Reference Manual*, Control Data Corporation Technical Publications Department, Arden Hills, Minn., 1971.
[11] D. Wedel, "Fortran for the Texas Instruments ACS System", *ACM Sigplan Notices*, Vol. 10 No. 3 (March 1975), pp. 119-132.
[12] R. G. Zwakenberg, "Vector Extensions to LRLTRAN", *Sigplan Notices*, Vol. 10, No. 3 (March 1975), pp. 77-86.
[13] G. Paul and M. Wilson, "An Introduction to VECTRAN and Its Use in Scientific Computing," *Proceedings of the 1978 LASL Workshop on Vector and Parallel Processors*, 1978, pp. 176-204.
[14] *Burroughs Scientific Processor Vector Fortran Specification* Defense, Space, and Special Systems Group, Burroughs Corporation, Paoli, Penn., 1978.
[15] *Fortran 8x, X3J3/S8 (X3.9-198x)*, American National Standards Institute, 1986.
[16] *Cray-1 Fortran (CFT) Reference Manual*, Pub. no. SR-0009, Cray Research, Inc., Mendota Heights, Minn., 1979.
[17] *Fortran 200 Version 1 Reference Manual*, Control Data Corporation Technical Publications Department, Arden Hills, Minn., 1982.
[18] K. Miura and K. Uchida, "FACOM Vector Processor VP-100/VP-200," *Proc. NATO Advanced Research Workshop on High-speed Computing*, Julich, W. Germany, Springer-Verlag, June 20-22, 1983.
[19] R. Scarborough and H. Kolsky, "A Vectorizing Fortran Compiler," *IBM Journal of R and D*, 30(2), March 1986, pp. 163-171.
[20] *FX/Fortran Language Manual*, Alliant Computer Systems Corporation, Acton, Mass., April, 1985.
[21] T. Watanabe, "Architecture and Performance of NEC supercomputer SX system," *Parallel Computing*, Vol. 5 (1987), pp. 247-255.
[22] D. Kuck, Y. Muraoka, and S. Chen, "On the Number of Operations Simultaneously Executable in Fortran-like Programs and their Resulting Speedup," *IEE Transactions on Computers*, Vol. C-21 no. 12 (Dec. 1972).
[23] D. Kuck, P. Budnick, S. Chen, E. Davis, Jr., J. Han, P. Kraska, D. Lawrie, Y. Muraoka, R. Strebendt, and R. Towe, "Measurements of Parallelism in Ordinary Fortran Programs," *IEEE Computer*, Vol. 7 no. 1 (Jan. 1974).
[24] D. Kuck, R. Kuhn, B. Leasure, and M. Wolfe, "The Structure of an Advanced Vectorizer for Pipelined Processors," *Fourth International Computer Software and Applications Conference*, October 1980.
[25] R. Allen and K. Kennedy, "PFC: A Program to convert Fortran to Parallel Form," Tech. report MASC-TR 82-6, Department of Mathematical Sciences, Rice University, March 1982.
[26] J. Dennis and E. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM*, Vol. 9 No. 3 (March 1966), pp. 143-155.
[27] *HEP Fortran 77 User's Guide*, Pub. no. 9000006, Delelcor, Inc., Aurora, Co. 1982.
[28] *Cray Computer Systems Technical Note: Multitasking User Guide*, Pub. no. SR-0222, Cray Research, Inc., Mendota Heights, Minn., Jan. 1985.
[29] "Notes from ETA Presentation," Second International Conference on Supercomputing, Santa Clara, CA., May 1987.
[30] E. Dijkstra, "Cooperating Sequential Processes," *Technical Report EWD-123*, Techological University, Eindhoven, The Netherlands, 1965, reprinted in [Genu86], pp. 43-112.
[31] A. Veidenbaum, "Compiler Optimizations and Architecture Design Issues for Multiprocessors," CSRD doc. no. 520, U of I Center for Supercomputing R and D, 1985.
[32] *Final Report – Numerical Aerodynamic Simulation Facility Feasibility Study*, Defense, Space, and Special Systems Group, Burroughs Corporation, Paoli, Penn., March 1979.
[33] D. Padua, "Multiprocessors: Discussion of Some Theoretical and Practical Problems", Ph.D. Thesis, report 79-9990, Dept. of Computer Science, Univ. of Illinois Urbana-Champaign, Oct. 1979.
[34] R. Cytron, "Doacross: Beyond vectorization for multiprocessors." *Proceedings 1986 International Conference on Parallel Processing*, Aug. 1986, pp. 836-844.
[35] C. Polychronopoulos, *On Program Restructuring, Scheduling, and Communication For Parallel Processor Systems*, CSRD doc. no. 595, U of I Center for Supercomputing R and D, August 1986.
[36] A. Osterhaug, *Guide to Parallel Programming on Sequent Computer Systems*, Sequent Computer Systems, Inc., Beaverton, Oregon, 1986.
[37] *FX/Series Architecture Manual*, Alliant Computer Systems Corporation, Acton, Mass., April, 1985.
[38] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, V. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proceedings 1985 International Conference on Parallel Processing*, Aug. 1985, pp. 764-771.
[39] *Encore Parallel Processing Guide*, Encore Computer Corporation, Marlborough, Mass., 1988.
[40] *Notes from the Argonne National Labs Parallel Computing Forum*, Argonne National Labs, Nov. 1987.
[41] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *IEEE Computer*, Vol, 19 no. 8 (Aug. 1986), pp. 26-34.
[42] *Programming Language Fortran ANSI x3.9-1978*, American National Standards Institute, 1978.
[43] Perry Emrath, "Xylem: An Operating System for the Cedar Multiprocessor", *IEEE Software*, Vol. 3 No. 4 (July 1985), pp. 30-37.
[44] C. Hoare, "Towards a Theory of Parallel Programming," *Operating System Techniques*, C. Hoare and R. Perrott (Eds.), Academic Press, London, 1972, pp. 61-71.
[45] C. Zhu and P. Yew, *A Scheme to Enforce Data Dependence on Large Multiprocessor Systems*, CSRD doc. no. 40, U of I Center for Supercomputing R and D, July 1984.

| System | Microtasking | | | | Data | Macrotasking | | | |
|---|---|---|---|---|---|---|---|---|---|
| | doall | doacross | high-level spreading | automatic parallelization | shared/private common | spawn | wait | locks | events |
| Alliant | no | yes | no | yes | no | no | no | no | no |
| BBN Butterfly – future plans | Doparallel,unlocked<br>Do ...<br>Continue | Doparallel,locked<br>Do ...<br>Begin Ordered<br>End Ordered | Begin Parallel<br>End Section | no | Shared Common<br>Common | no | no | yes | no |
| Cray | yes | no | yes | no | Common<br>Task Common | yes | yes | yes | yes |
| Encore Parallel Fortran | yes | doall<br>send/wait<br>end doall | Parallel<br>End Parallel | yes | Shared<br>Private<br>Volitile | no | wait and barrier | yes | yes |
| IBM Parallel Fortran PRPQ – future plans | Parallel loop<br>Continue | no | Parallel Case<br>End Case | yes | yes | yes | yes | unknown | unknown |
| Sequent | no | yes | no | yes | shared by compiler option | yes | yes | yes | no |

Table 1. Comparison of Multiprocessor Tasking Systems