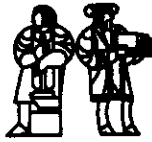
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

# Future Scientific Programming on Parallel Machines

Computation Structures Group Memo 272
March 1987
Revised February 1988

**Arvind**
**Laboratory for Computer Science**
**Massachusetts Institute of Technology**
**Cambridge, Massachusetts**

**Kattamuri Ekanadham**
**IBM T.J. Watson Research Center**
**Yorktown Heights, New York**

Source File = COVER.MSS.7, Last updated 18-FEB-88 at 1:29 PM

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

# Future Scientific Programming on Parallel Machines

Arvind
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts

Kattamuri Ekanadham
IBM T.J.Watson Research Center
Yorktown Heights, New York

Proposed running head: *Future Scientific Programming*

Mailing address for proofs:
Kattamuri Ekanadham
IBM T.J.Watson Research Center
P.O.Box 704
Yorktown Heights, New York 10598.

*ABSTRACT:*

*A language for large scientific applications should facilitate encoding and debugging of programs at the highest level possible. At the same time it should facilitate generation of efficient code for parallel machines. Often these two requirements are conflicting, and trade-offs must be made. Functional and other declarative languages offer relief on both counts. The use of higher-order functions, especially in curried forms, can raise the level of programming dramatically. In addition, such languages often have straightforward operational semantics, thereby providing tremendous opportunities for parallel execution. Programs written in declarative languages thus eliminate the problem of "detecting parallelism". This paper illustrates programming in one such language, Id Nouveau, and contrasts it with programming in Fortran. Using an excerpt from an application known as Simple, it is shown how a program can be composed in Id Nouveau from small functions that directly relate to the mathematical and physical concepts of the problem. The difficulty of expressing these concepts in Fortran is discussed. Finally, it is shown that by performing simple transformations, such as in-line substitution of functions, the resulting Id Nouveau code becomes as efficient as an equivalent Fortran program written to run efficiently on a parallel machine.*

Special symbols and types used

relational operators such as $< \leq > \geq$
curly braces $\{ \}$
ellipsis as in ...
right arrow $\rightarrow$
vector notation as in $\vec{v}$
Greek letters: $\alpha$ $\rho$ $\delta$ $\varepsilon$ $\lambda$ $\theta$
line integral symbol $\oint$
implication symbol $\Rightarrow$
parallel symbol []
primed alphabet as in $e'$
italic and boldface fonts at many places as indicated in the text

# 1. Introduction

The world's fastest computers have traditionally been used first in scientific computing. Yet, the level of programming, as represented in typical commercial scientific packages or in the large codes written in national laboratories, is remarkably low. The usual excuse for this situation is that high performance is the name of the game in scientific computing, and the programmer must exercise fine-grain control over the execution environment to tailor the program to a given machine. Because all important programs need to be modified, a program during its lifetime (often spanning more than a decade) gets so complicated that the original algorithm is obscured and the efficiency of the program becomes more a question of faith than reason.

These observations would be easy to ignore if the development of new codes employed modern computer science techniques. In fact, the advent of parallel high-speed computing seems to be causing a regression in programming. Further, the users are being told that these new software problems are a natural consequence of the need to develop new algorithms for parallel machines. There can be no argument that in due course more suitable algorithms will emerge for parallel computers. However, this does not imply that algorithms already in use are necessarily inappropriate for parallel machines. We believe that the root of the problem is the use of Fortran to implement these algorithms.

Two interrelated techniques for the development of software for purported general-purpose parallel machines are identifiable in the present commercial environment. Both are geared to "dusty decks" of Fortran, though they can be applied to new Fortran programs as well. One requires the programmer to *annotate* the program to indicate the opportunities for parallel execution, and the other requires the compiler to *detect* the program fragments that can be executed in parallel [17]. A typical annotation is "Do-all" to execute all iterations of a Do-loop in parallel. Similar annotations are

usually available for concurrently executing the statements of a block. Parallel execution requires synchronization which often has to be coded explicitly. Machine-specific concerns often require classification of variables as local or global, cacheable or non-cacheable, ultimately making the program more and more obscure. Since annotations interact with the storage model of the language, the meaning of programs changes in subtle ways. Annotations are rarely robust. For example, nested "Do-all's" may not be permitted because Fortran does not have dynamic storage allocation. Worst of all, wrong annotations (such as specifying a variable as global when it should have been local or specifiying a loop to execute in parallel when there is some undetected dependency etc.) invariably make the behavior of the program *non-deterministic*, that is, time- and machine configuration-dependent, thereby creating a nightmare during debugging.

Detection of parallelism by a compiler is very desirable from a user's point of view [17]. However, even the most sophisticated techniques to detect parallelism trip on trivial impediments such as conditionals, function calls, input/output statements, etc., and fail to detect most of the parallelism present in a program. Some parallelizing compilers provide feedback to the user regarding where the compiler had difficulty in deciding about parallel execution (see discussion of the Fujitsu Fortran compiler in [11]). Based on these suggestions, a programmer can often restructure the source code to aid the detection of parallelism. In principle, there is no reason why such a compiler cannot also accept annotations to improve the quality of generated code. Useful as this methodology is for dealing with "dusty decks", it only confirms the lack of expressivity of the source language.

Perhaps one reason why languages like Fortran still dominate the scene is the control placed in the hands of the programmer to extract the last ounce of efficiency from a system. Despite the pain attached to its use, Fortran use is likely to continue until a viable alternative demonstrates comparable performance.

Functional and declarative languages are said to offer many advantages in this context [5,20]. Their declarative nature eliminates the overspecification of the order of evaluation. Their operational semantics when expressed in terms of rewrite rules automatically exposes the parallelism present in a program. Functional languages with higher-order functions elevate the level of programming so that abstractions can be built closer to the concepts in the problem domain. Functional programs are easier to reason about because their output is determinate, that is, independent of the order of evaluation. However, functional languages traditionally have lacked good facilities for manipulating arrays and matrices; simulating such structures using traditional functional data structures often results in excessive storage demand or code which is unnecessarily sequential [3]. A declarative language called *Id Nouveau* [15,4] has been proposed as a solution to some of these problems. *Id Nouveau* (simply *Id* from now on) is a functional language augmented with a novel array-like data structure called *I-structures.* This language is being used to program the M.I.T. Tagged-Token Dataflow machine. In addition to the requirement of generating good code, *Id* also embodies the advantage of declarative programming, that is, code which is clear, concise and easy to understand and reason about. The objective of this paper is to show, by a realistic example, the expressive power of this language, as well as its ability to expose the parallelism in a specification in a natural manner, without requiring annotations.

In Section 2, we describe the example which has been excerpted from a hydrodynamics simulation program known as the Simple code [6]. In Sections 3.2 and 3.3, we describe some matrix abstractions which are useful for writing the Simple code and, we believe, many other codes. In order to make the paper self-contained for readers unfamiliar with functional programming, basic notions of functions and I-structures are introduced in Section 3.1 using the *Id* syntax. The reader is referred to [15,3] for a more detailed treatment of *Id* and I-structures. Section 3.4 presents the *Id* solution to the main example. A complete *Id* program for Simple has been developed by the authors and is reported in [8]. In Section 4, we point out the difficulties in imitating the declarative style of programming in Fortran. In Section 5, we show that, with "in-line substitution"

of functions, the performance of the *Id* program of Section 3.4 becomes comparable to that of an equivalent Fortran program. Section 6 discusses some storage efficiency issues. Finally, Section 7 presents our conclusions.

# 2. An Example: Problem Description

Our example problem is an excerpt from a hydrodynamics and heat conduction simulation program known as the Simple code [6]. The document [6], along with the associated Fortran program, was developed as a benchmark to evaluate various high performance machines and compilers. The problem is to simulate the behavior of a fluid in a sphere, using a Lagrangian formulation of equations. To simplify the problem, only a semi-circular cross-sectional area (see Figure 1) is considered for simulation. The area is divided into parcels by radial and axial lines as shown. Each parcel is delimited by four corners as illustrated in Figure 1. The corners are called *nodes*. Each region enclosed by 4 nodes is called a *zone* (illustrated by the shaded area in Figure 1). In the Lagrangian formulation, the nodes are identified by mapping them onto a 2-dimensional logical grid, in which the grid points have coordinates $(k, l)$ for some $kmin \leq k \leq kmax$, $lmin \leq l \leq lmax$. The product, $kmax * lmax$, is often referred to as the *grid size* of the problem. The following quantities are considered in the simulation.

1. $\vec{v}$ : velocity components, u,w, of node (k,l) in the R-Z plane.
2. $\vec{x}$ : coordinates, r,z, of node (k,l) in the R-Z plane.
3. $\alpha$ : Area of zone (k,l).
4. s : Volume of revolution per radian of zone (k,l).
5. $\rho$ : Density of zone (k,l).
6. p : Pressure of zone (k,l).
7. q : Artificial Viscosity of zone (k,l).
8. $\varepsilon$ : Energy within zone (k,l).
9. $\theta$ : Temperature of zone (k,l).

Each quantity is stored as a matrix of values, indexed by the indices of a node. For

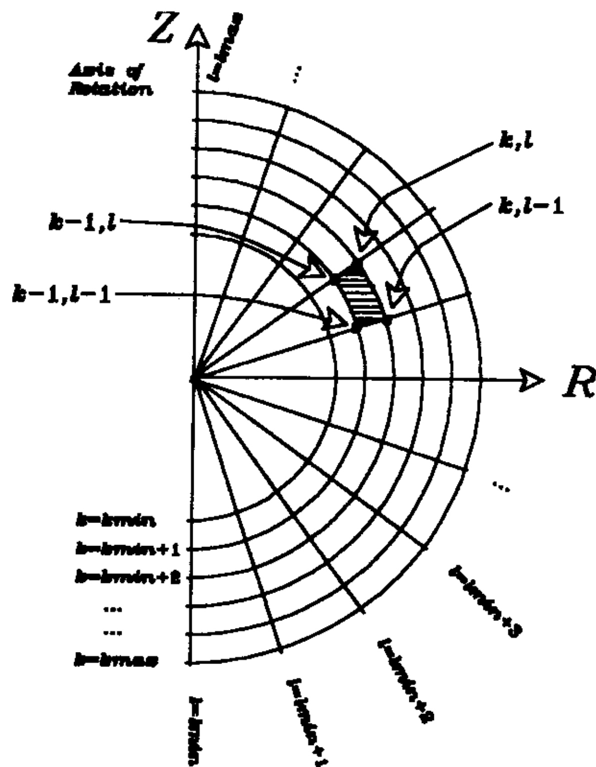



**Figure 1: Fluid parcels in a 2-dimensional cross-section**

each time period of $\delta t$, a new matrix is computed for each quantity using appropriate values from the preceding time step. Note that the first two quantities are associated with nodes and the remaining are associated with zones.

In order to incorporate appropriate boundary conditions, a fictitious layer of zones, called *ghost zones*, is added to the cross section, as depicted by the shaded area in Figure 2. Rows *kmin-1* and *kmax+1* and columns *lmin-1* and *lmax+1* contain the boundary nodes. Changes in the quantities associated with ghost zones are governed by the desired boundary conditions. Each quantity is associated permanently with a criterion to determine its boundary value. At each time step new boundary values for a quantity are computed according to the associated criterion.

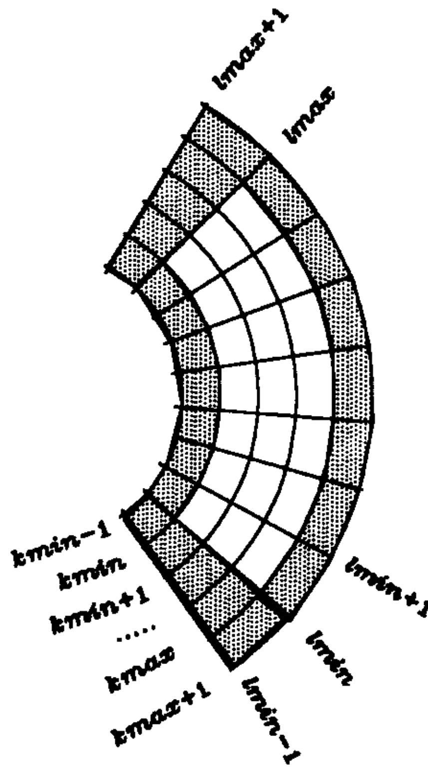



**Figure 2: Ghost zones added for boundary conditions**

For convenience, the following nomenclature is adopted to refer to the neighboring nodes and zones of a node. The neighboring nodes are named *north, south, east, west.* For example, in Figure 3 the north neighbor of node $(k,l)$ is $(k\text{-}1,l)$ and the west neighbor is the node $(k,l\text{-}1)$. The 4 zones around a node $(k,l)$ are named $A$, $B$, $C$, $D$ in a counterclockwise manner as shown in Figure 3. A zone is identified by specifying its *southeast* corner node. Thus, for example, the B zone of node $(k,l)$ is referred by indices $(k+1,l)$.

In this paper, we will discuss only the computation of the new velocity at each node. The velocity is not defined at the boundary nodes and hence our example deals only with interior nodes. The velocity at each node is determined by first computing the

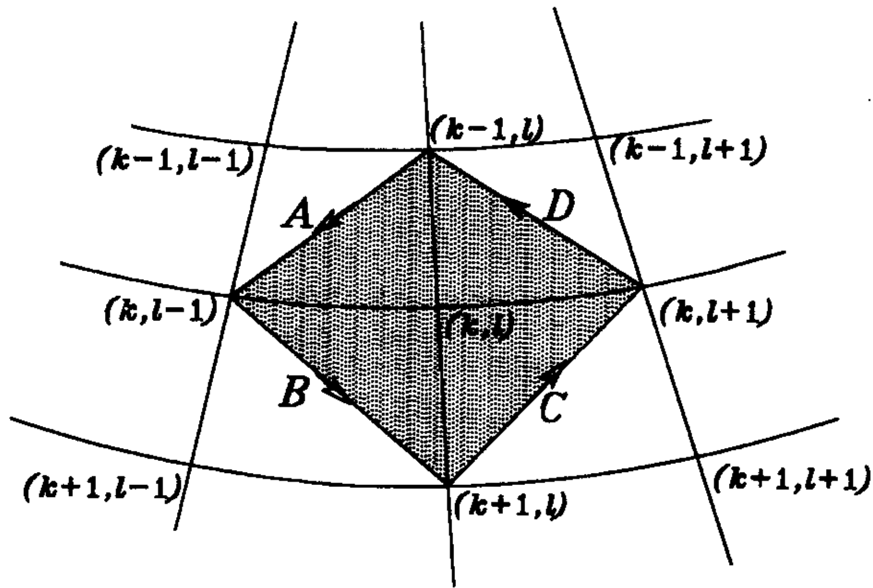



**Figure 3: Velocity computation at a node**

acceleration at that node during the time step and incrementing the old velocity by the product of time and acceleration. The acceleration is obtained from the equation for conservation of momentum. After some simplifying algebra, the acceleration is given by

$$\frac{d\vec{v}}{dt} = \left[\frac{-\oint p\,dz - \oint q\,dz}{nodal_mass}, \frac{\oint p\,dr + \oint q\,dr}{nodal_mass}\right] \qquad \text{(A)}$$

In the numerator, the line integrals are to be taken over the boundary line of the shaded region around the node *(k,l)* shown in Figure 3. The line has four segments, one in each of the neighboring zones. While integrating, it is assumed that each zone quantity is constant along the line segment within that zone. Thus, for example, suppose we are integrating p with respect to z, along the line in zone A in Figure 3. Its value is given by the product of p in zone A and the difference of z between the two end

points of line segment A, that is $p[k,l] * (z[k,l-1] - z[k-1,l])$. The denominator, *nodal_mass*, is the mass of the shaded area around the node *(k,l)* in Figure 3. This is approximated as one half of the total mass of the 4 zones meeting at node *(k,l)*. The mass of a *zone* is the product of its density and area.

# 3. Programming in a Declarative Language

In this section, we discuss how some basic abstractions to manipulate matrices are programmed in *Id*. We build these abstractions gradually, by way of examples, while introducing *Id*. The introduction to *Id* is informal and written with the purpose of helping the reader to understand the main example to be presented later. However, care is taken to explain how the operational semantics of the language automatically allows concurrency in loop iterations and in data structure operations.

## 3.1. Functions, I-structures and Parallel Execution

***Functions:*** A function to add two numbers may be written in *Id* as follows:

$$add\ (i,j) = i+j\ ; \qquad (1)$$

This function can be used to add, say *2* and *3*, by writing *add (2,3)*. In a declarative language such as *Id*, the operational meaning of a program can be explained in terms of *rewrite rules* because every definition can be read as a rewrite rule. A function application is rewritten as or *reduced* to the right-hand side of its definition by substituting for the parameters. For example, *add (2,3)* is reduced to $2+3$ by substituting *2* for $i$ and *3* for $j$ in the right-hand side of the definition of *add*. This process of reduction is repeated until we arrive at a form that cannot be reduced any more; this is deemed the result of the computation. Assuming natural rewrite rules for primitive functions, such as $+$, we can reduce $2+3$ to *5*. We could also have written the function to add two numbers as follows:

$$c_add \; i \; j \; = \; i+j; \tag{2}$$

and applied it by writing *c_add 2 3*. It is customary in functional languages to write a function application as *f a* rather than *f(a)*. Further, to reduce the number of parentheses in expressions, function application is considered to be "left associative" by convention. Thus, the following expressions are equivalent:

*f a b*      *(f a) b*      *((f a) b)*      *f (a) b*

and none of them is equal to *f (a,b)*. In *Id*, the use of parantheses is only to disambiguate an expression.

The difference between *add* and *c_add* is subtle and important. The *add* function takes one argument which must be a 2-*tuple*. (A tuple is a data structure, as will be explained shortly.) The rewrite rule for *c_add*, on the other hand, can be applied only when *c_add* is followed by two arguments. In functional language parlance, *arity* of *c_add* is 2, while *arity* of *add* is 1. The significance of this difference becomes apparent when we consider the following *binding*, that is, an association of a name with an expression:

$$successor \; = \; c_add \; 1; \tag{3}$$

*Successor* or equivalently *c_add 1* is a function that adds one to a number, that is, *successor n* will behave like *c_add 1 n*. Expressions such as *c_add 1* are called *partial applications* because *c_add* is being applied to fewer arguments than are needed to "fire" the corresponding rewrite rule. Thus, partial applications remain irreducible until a sufficient number of arguments is supplied. Such expressions represent higher-order functions, which are essential for writing abstract programs. Notice, it will make no sense to write *add 1* because *add* expects a tuple as an argument. In functional languages, *c_add* is called the "curried" version of *add* after the famous logician Haskell B. Curry.

***Tuple structures:*** A *tuple* is a data structure. Commas are used as tuple constructors in *Id*. Thus, *(1,n),(1,n)* is a 2-tuple of two 2-tuples. We may represent the bounds of an array as a 2-tuple of integers and the bounds of a matrix as a 2-tuple of 2-tuples. Thus, *grid* in

$$grid\ \ n \ = \ ((1, n), (1, n)); \qquad (4)$$

is a function that takes a number $n$ as an argument and returns a value that may be used to express the bounds for an $n \times n$ matrix.

A function in *Id* can return only one value. As illustrated below, a tuple can be used to package results when a function needs to return multiple results:

$$\begin{aligned} north\ \ (i,j) &= (i\text{-}1,j); \\ west\ \ (i,j) &= (i,j\text{-}1); \end{aligned} \qquad (5)$$

Function *north* takes the indices of an element in a matrix and returns the indices of its north neighbor. Similarly, the function *west* returns the neighbor on the west side. As we shall see later, such abstractions are useful when calculations have to be performed for each node of a grid using values at neighboring nodes. For example, using these abstractions we can write *(north node)* where *node* is *(i,j)*, instead of the subscript expression *(i-1,j)*. Such abstractions significantly reduce the likelihood of erroneous subscripts in programming.

***Blocks:*** A *block* expression in *Id* is a set of bindings followed by the key word ***in*** and a return expression. It provides a convenient way to share the computation of common subexpressions, as is shown in the following definition containing a block.

$$
\begin{array}{ll}
\textit{acceleration node} = & (6) \\
\{ \quad d \;=\; \textit{nodal_mass node;} & \\
\quad n1 \;=\; -(\textit{line_integral p z node}) - (\textit{line_integral q z node}); & \\
\quad n2 \;=\; (\textit{line_integral p r node}) + (\textit{line_integral q r node}); & \\
\textbf{in} \quad n1 / d, \;\; n2 / d \}; &
\end{array}
$$

For now, we will assume that *line_integral* and *nodal_mass* are known functions and *p, q, r, z* are known constants. This block contains three bindings, one each for *d, n1* and *n2*, and returns a 2-tuple. The use of identifier *d* in the return expression allows the computation of the subexpression *(nodal_mass node)* to be shared. *Id* uses "lexical scoping" rules. Thus, a name that is bound in a block, *e.g. n1*, is invisible outside the block. Similarly, a name used in a block, but not defined in that block, *e.g. p*, imports the value from a binding to that name in the nearest enclosing block.

We explain the execution of block expressions in terms of rewrite rules. (A precise operational semantics for *Id*, using dataflow graphs, is given in [19].) In the following, we use the notation $e \Rightarrow e' \;\; [] \;\; \textit{form1} \;\; [] \;\; \textit{form2} \ldots$ to show that $e'$ is obtained by rewriting some subexpressions in *e*, while the computations in *form1, form2,* ... are started concurrently. In the following, let *node* denote some *value* such as *(2,3)*. Then the function application *acceleration node* can be reduced by substituting the node value in the body of the function:

$$
\begin{array}{l}
\textit{acceleration node} \\
\Rightarrow \{ \quad d \;=\; \textit{nodal_mass node;} \\
\qquad n1 \;=\; -(\textit{line_integral p z node}) - (\textit{line_integral q z node}); \\
\qquad n2 \;=\; (\textit{line_integral p r node}) + (\textit{line_integral q r node}); \\
\quad \textbf{in} \quad n1 / d, \;\; n2 / d \; \}
\end{array}
$$

A block is reduced to its return expression and the computations in all the bindings of the block are initiated concurrently as shown below:

$$
\begin{array}{lll}
\Rightarrow & & n1 \,/\, d,\;\; n2 \,/\, d \\
& [] \; d & = \; \textit{nodal_mass node} \\
& [] \; n1 & = \; -(\textit{line_integral p z node}) - (\textit{line_integral q z node}) \\
& [] \; n2 & = \; \quad (\textit{line_integral p r node}) + (\textit{line_integral q r node})
\end{array}
$$

Note that the order in which bindings appear in the original expression is immaterial. Assuming the arities of *nodal_mass* and *line_integral* are satisfied, we may rewrite all the 5 function calls.

$$
\begin{array}{lll}
\Rightarrow & & n1 \,/\, d,\;\; n2 \,/\, d \\
& [] \; d & = \; (\textit{...body of nodal_mass ...}) \\
& [] \; n1 & = \; -(\textit{...body with p z etc. ...}) - (\textit{...body with q z etc. ...}) \\
& [] \; n2 & = \; \quad (\textit{...body with p r etc. ...}) + (\textit{...body with q r etc. ...})
\end{array}
$$

In *Id*, substitution for an identifier is performed only when the right-hand side of the corresponding binding reduces to a *value*, that is, a number, boolean, tuple or partial application [4]. The binding may be ignored after it has been substituted everywhere. Suppose we rewrite expressions in the body of *nodal_mass* until it reduces to a value designated by $v$. Substituting $v$ for $d$, we get

$$
\begin{array}{lll}
\Rightarrow & & n1 \,/\, v,\;\; n2 \,/\, v \\
& [] \; n1 & = \; -(\textit{...body with p z etc. ...}) - (\textit{...body with q z etc. ...}) \\
& [] \; n2 & = \; \quad (\textit{...body with p r etc. ...}) + (\textit{...body with q r etc. ...})
\end{array}
$$

Similarly, the four *line_integrals* may be reduced to values $v1$, $v2$, $v3$ and $v4$, respectively, to produce

$$\Rightarrow \quad \begin{array}{l} n1/v,\ \ n2/v \\ [\,]\ \ n1 = -v1 - v2 \\ [\,]\ \ n2 = \ \ v3 + v4 \end{array} \quad \Rightarrow \quad \begin{array}{l} n1/v,\ \ n2/v \\ [\,]\ \ n1 = v5 \\ [\,]\ \ n2 = v6 \end{array} \quad \Rightarrow \ v5/v,\ \ v6/v \ \Rightarrow \ v7,\ \ v8$$

where $v5$ through $v8$ have obvious meanings.

At this stage, the expression cannot be reduced any further and we say that the answer has been found. (We will rely on the reader's intuition to figure out when a tuple represents a value as opposed to an unevaluated tuple.) It is worth noting that at several steps of this reduction sequence, we had a choice of subexpressions to reduce. Such a choice represents an opportunity for parallel execution. The most wonderful property of *Id* (and other functional languages) is that the order in which we reduce reducible expressions has no effect on the final answer. This property, known as the *Church-Rosser property*, guarantees the *determinacy* of computation. When Fortran programs are parallelized by annotations, determinacy is not guaranteed by the language. It is the responsibility of the programmer to make sure that the annotated synchronizations preserve the determinacy of the computation. It is the indeterminacy due to erroneous annotations that creates nightmares while getting a parallelized Fortran program to work. The same criticism applies to other imperative languages such as Lisp, C, Pascal *etc.*

Very frequently, curried functions are defined implicitly. For example, suppose definition (6) appears in a block where $p, q, r, z$ are defined. The values of these variables can be supplied for computing the *acceleration*, by including them as additional arguments in the definition of *acceleration*. The use of such implicit arguments is permitted in *Id* for programming convenience; the compiler automatically transforms the programs (using a technique known as *λ-lifting* [9]) into equivalent programs, where

all arguments are explicit. For example, the compiler will transform definition (6) as follows, where *acceleration'* is a new function of arity 5 and *acceleration* is its curried form.

```
acceleration' p q r z node =
  {   d  = nodal_mass node;
      n1 = - (line_integral p z node) - (line_integral q z node);
      n2 =   (line_integral p r node) + (line_integral q r node);
  in  n1 / d,  n2 / d};

acceleration = acceleration' p q r z;
```

***I-structures:*** Now we introduce I-structures, a novel data structuring facility which has been designed primarily to facilitate parallel programming. One can think of an I-structure as a special kind of array which is allocated at run time and whose elements can be written (that is, defined) no more than once. It is allocated by the expression ***array*** *(l,u)*, which allocates and returns an "empty" array with index bounds *l* and *u*. We will show the execution of this primitive by the following rewrite rule:

$$\textbf{\textit{array}}\ (l, u) \Rightarrow \ < al, al+1, \ldots, au >$$

where each *ai* represents a memory location and $<al, al+1, \ldots, au>$ represents an I-structure *value*.[1] I-structure elements can contain anything, including other I-structures. Thus, one can define a matrix as an array of arrays. However, for the sake of efficiency, we also provide matrix as a primitive data structure which may be allocated using the expression ***matrix*** *((l1,u1),(l2,u2))*. A component of an I-structure *A* may be assigned

---

1 We would like to point out that this rewrite rule takes us out of the scope of functional languages, because variables appearing on the right-hand side are not taken from the variables on the left-hand side. However, such rules are common and necessary for defining the operational semantics of logic languages.

(written) no more than once using a "constraint statement": $A[i] = v$. It is a run-time error to write into an I-structure location more than once - the entire program is considered to be in error. The *i-th* component of an I-structure $A$ is selected by writing the expression $A[i]$. If $A$ is the structure $<a1, ...au>$, then the selection $A[i]$ first computes the index of the array. This is indicated by the reduction of $A[i]$ to the name *ai*. Then the location is read. The reading is indicated by the reduction of *ai* to a value. Thus, the expression $A[i]$ returns a value only after the location has been assigned a value. It is important to realize that no test for the "emptiness" of an element is provided for the programmer. This restriction is necessary to preserve the determinacy of the language.

The operational view of I-structures given above can also be described in terms of rewrite rules, as illustrated by the following example. First, we reduce the block expression:

```
{    A = array (1, 10);              =>    A [ 5 ]
     A [ 5 ] = A [ 1 ] + 4;              [] A = array (1, 10)
     A [ 1 ] = 3;                        [] A [ 5 ] = A [ 1 ] + 4
in   A [ 5 ] }                           [] A [ 1 ] = 3
```

After the storage allocation for the array is done, $A$ becomes a value of the form $<a1, a2, ..., a10>$, and, hence, it can be substituted:

```
=>      < a1, a2, ..., a10 > [ 5 ]
    []  < a1, a2, ..., a10 > [ 5 ] = < a1, a2, ..., a10 > [ 1 ] + 4
    []  < a1, a2, ..., a10 > [ 1 ] = 3
```

```
=>      a5
    []  a5 = a1 + 4
    []  a1 = 3
```

At this stage only *a1* can be substituted because *a5* has not yet become a *value.*

$$\Rightarrow \quad \begin{array}{l} a5 \\ [\,]\ a5 = 3 + 4 \end{array} \qquad \Rightarrow \quad \begin{array}{l} a5 \\ [\,]\ a5 = 7 \end{array} \qquad \Rightarrow \quad 7$$

The crucial point in understanding these rewrite rules is that one is allowed to substitute for an identifier only after the right-hand side of the equation associated with the identifier has been reduced to a value. If the right-hand side never becomes a value (that is, is never defined), then no substitution for the identifier is ever done and the "read" operation will take forever!

***Loops:*** A *for-loop* expression in *Id* is a set of bindings to be executed repeatedly for a specified number of times. All iterations execute concurrently. The following example illustrates a for-loop.

```
{    A = array  1,10 ;                                    (7)
     {for i from 1 to 10 do
          A [ i ] = i * i }
in   A }
```

First, the block expression is reduced. Then the loop is reduced to a set of parallel iterations, one for each value of the control variable. Each iteration is a copy of the body of the loop in which the corresponding value is substituted for the control variable, as shown below:

```
⇒      A
    [] A = array 1, 10
    [] {for i from 1 to 10 do
          A [ i ] = i * i }
```

```
⇒      A
    [] A = array 1, 10
    [] A [ 1 ]  = 1 * 1
    [] ...
    [] A [ 10 ] = 10 * 10
```

Suppose the storage allocation for the array is completed first. The name $A$ is then replaced by its array descriptor, *i.e.* the value $<a1, a2, \ldots, a10>$. Since $A$ is the result expression, the answer becomes available even though the values of the array elements are still being computed. Each of the index selections is replaced by the corresponding names, as discussed earlier. Finally, the individual values are stored into the corresponding elements.

```
⇒      < a1, a2, ..., a10 >
    [] < a1, a2, ..., a10 > [ 1 ]  = 1 * 1
    [] ...
    [] < a1, a2, ..., a10 > [ 10 ] = 10 * 10
```

```
⇒      < a1, a2, ..., a10 >
    [] a1  = 1
    [] ...
    [] a10 = 100
```

It should be noted that the iterations are executed concurrently regardless of the nature of the computation. For example, in the following loop, although the multiplications proceed concurrently, the additions and the store operations take place sequentially as dictated by data dependencies.

```
{     A = array 0, 10 ;
      A [ 0 ] = 0;
      {for i from 1 to 10 do
           A [ i ] = A [ i-1 ] + i * i }
in    A }
```

```
⇒ ... ⇒     < a0, a1, ..., a10 >
          [] a0  = 0
          [] a1  = a0 + 1 * 1
          [] ...
          [] a10 = a9 + 10 * 10
```

As another variation, consider the following example in which the additions and store operations take place in the reverse order. This shows that the loop specification is declarative in the sense that *for i from 1 to 10* specifies the *set* of indices and not the order of their traversal. The order, if any, is governed by data dependencies.

```
{   A = array 1,11 ;                     =>  ...  =>      < a1, a2, ..., a11 >
    A [ 11 ] = 0;                                      [] a11 = 0
    {for i from 1 to 10 do                             [] a1  = a2 + 1 * 1
        A [ i ] = A [ i+1 ] + i*i }                    [] ...
in  A }                                                [] a10 = a11 + 10 * 10
```

Loops may also return a value. The set of bindings in such a loop is followed by the keyword ***finally*** and a return expression. All the iterations and the return expression are initiated concurrently. The following example illustrates the reduction of a set of elements to their sum. The key word ***next*** indicates that the value computed in the *i-th* iteration for the right-hand side of the binding is bound to the name $s$ in the $i+1^{st}$ iteration. For clear exposition of this, the name $s$ used in iteration $i$ is renamed as $s_i$ and the following rewrite rules show how the recurrence is computed sequentially. The key word ***finally*** indicates that the value returned by the loop expression is the value assigned to $s$ in the last iteration.

```
{   s = 0 ;                              =>  ...  =>      s10
    in  {for i from 1 to 10 do                         [] s = 0
           next s = s + i * i                          [] s1 = s + 1 * 1
        finally  s } }                                 [] s2 = s1 + 2 * 2
                                                       [] ...
                                                       [] s10 = s9 + 10 * 10
```

## 3.2. *Abstractions to Manipulate Arrays and Matrices*

The array $A$ in definition (7) can be thought of as an efficient representation for the function *square* $i = i * i$ over the finite domain *(1,10)*. Instead of computing *square* $i$ each time it is invoked, the values of *square* $i$ for all $i$ in the domain are computed and stored in memory. Thus, $A$ acts like a "cache" for the function *square*.[2] The following abstractions emphasize this viewpoint.

```
make_array (l, u) generate =                                    (8)
    {   A = array (l, u);
        {for i from l to u do
            A [ i ] = generate i }
    in  A};

make_matrix ((l1, u1), (l2, u2)) generate =                     (9)
    {   A = matrix ((l1, u1), (l2, u2));
        {for i from l1 to u1 do
          {for j from l2 to u2 do
            A [ i,j ] = generate (i,j) }}
    in  A};

A = make_array (1, 50) successor                                (10)
B = make_matrix (grid 50) add                                   (11)
```

In the two definitions (8) and (9), *generate* is the function that generates the elements

---

2 It would be nice if a language permitted the use of structures and functions interchangeably. However, this might introduce implementation inefficiencies and *Id* does not permit this for now.

of the corresponding structure and its finite domain is the set of all indices within the dimensions of the structure. For example, definition (10) builds a 50-element array in which the *i-th* element has value $i+1$. Similarly, definition (11) builds a $50 \times 50$ matrix, (using the *grid* function defined in (4)), in which the value of each element is the sum of the corresponding indices (using the *add* function defined in (1)).

An abstraction should enable us to model the essence of a concept so that it can be used in different circumstances. The robustness of an abstraction depends upon the flexibility with which the abstraction lends itself to a variety of uses. We will illustrate the robustness of the *make_array* and *make_matrix* abstractions defined above. We will show that these abstractions do not depend upon the type of the generating function, and that they expose all the parallelism present, irrespective of whether the computations of different elements are independent of each other or not.

***Arbitrary generating functions:*** In definition (8), *make_array* expects a *generator* function whose type is

$$integer \rightarrow anytype$$

where *anytype* represents the type of array elements. Similarly, a *generator* for *make_matrix* must have the type

$$integer \times integer \rightarrow anytype$$

Thus, the *make_matrix* abstraction works equally well with generator *add* (see definition (11)) which returns an integer, and generator *velocity* (given below) which returns a 2-tuple.

```
velocity node =                                                    (12)
    {   au, aw  =  acceleration node;
        u, w    =  Old_V [ node ] ;
    in  u + au * time,  w + aw * time };

V  =  make_matrix (grid 50) velocity;                              (13)
```

As an aside, we point out that the function *velocity* computes the velocity components in the two dimensions at a point specified by *node*. *Acceleration* is the function defined in (6), which gives the component accelerations in the two dimensions at a given *node*. *Old_V* is the old velocity matrix. The new velocity is obtained by incrementing the old velocity by the product of time and acceleration.

Another dimension of flexibility is that the generator function can be computed at run time. We can illustrate this using function *c_add* (definition (2)) whose type is:

$$integer \rightarrow (integer \rightarrow integer)$$

If *c_add* is applied to *n* then the result would be a function with type

$$integer \rightarrow integer$$

Thus, it makes sense to use *(c_add n)* as a generator for the make-array abstraction as shown below:

```
make_array (1, 50) (c_add 1)
```

This creates an array identical to that created in definition (10). We leave it to the reader to figure out the elements (and their types) of the following array.

*make_array (1, 50) c_add*

***Parallelism is dictated by data dependencies:*** Note that for both the abstractions (8) and (9), the structure, *i.e.* its storage descriptor, is returned as soon as the storage is allocated; the binding of the elements takes place in parallel. The implementation must guarantee that any premature accesses are synchronized.[3] Furthermore, all the iterations execute concurrently, unless constrained by data dependencies. For instance, in definitions (10) and (11) all the iterations are independent and hence will execute in parallel. On the other hand, consider the following example in which the velocity is defined as constant 55.5 at the north and west boundaries of the matrix and it is propagated along a wavefront - so that the velocity at a node is the sum of the velocities of its north and west neighbors:

```
rec_velocity V (i,j) =                                      (14)
    if i==1 or j==1 then 55.5
    else V[ north (i,j) ] + V[ west (i,j) ]

V = make_matrix (grid 50) (rec_velocity V);                 (15)
```

where "= =" is the relational operator for equality, to avoid overloading of the symbol "=". Definition (15) is recursive, as the name *V* is used in its own definition. However, the abstraction still works because the matrix is returned as soon as the storage is allocated. To illustrate this, we show a partial reduction sequence to compute V:

3 The I-structure implementation in the MIT Tagged-Token Dataflow Architecture uses tagged storage and deferred read lists for this purpose. For details see [2].

```
=>      V
   [] V = make_matrix (grid 50) (rec_velocity V);

=>      V
   [] V = {  A = matrix ((1,50),(1,50));
             {for i from 1 to 50 do
               {for j from 1 to 50 do
                  A [ i,j ]  = rec_velocity V (i,j)}}
          in  A};
```

Now replacing array $A$ by its value $<a_{1,1}, ... a_{50,50}>$ and substituting for the application of the function *rec_velocity* we get

```
=>      < a_1,1, ... a_50,50 >
   [] {for i from 1 to 50 do
        {for j from 1 to 50 do
           <a_1,1, ...a_50,50> [ i,j ]  = if i==1 or j==1 then 55.5
           else < a_1,1, ... a_50,50 > [ north (i,j) ] + < a_1,1, ... a_50,50 > [ west (i,j) ] }}
```

Now, one can see how the wavefront parallelism is unfolded as the velocities along each diagonal will be computed in parallel. The fact that the *make_matrix* abstraction is defined without regard to all these nuances of the function *generate* and that it automatically unravels all the parallelism present, no matter how the function *generate* is structured, shows that programming with such abstractions is highly desirable. Later, we will show how Fortran programmers go through all sorts of contortions to bring out this wavefront parallelism.

## 3.3. *Some More Matrix Abstractions*

We would like to introduce some more matrix-related abstractions which lead not only to clear programming style but also increased efficiency in execution.

***Make_2_matrices:*** Given a generating function that produces a 2-tuple, the following function defines a pair of matrices instead of a matrix of pairs:

```
make_2_matrices ((l1, u1), (l2, u2)) generate =                    (16)
    {   A = matrix ((l1, u1), (l2, u2));
        B = matrix ((l1, u1), (l2, u2));
        {for i from l1 to u1 do
          {for j from l2 to u2 do
            A [ i,j ] , B [ i,j ]  =  generate  (i,j) }}
    in  A, B};
```

For example, *velocity* of definition (12) is a function that returns a pair of values and (13) creates a matrix of pairs. The following creates a pair of matrices:

```
U, W = make_2_matrices (grid 50) velocity;
```

Sometimes it is convenient to maintain two matrices because there may be separate computations dealing with each velocity component. If a matrix of pairs were used, the components would have to be selected each time, thus causing some structure-accessing overheads. The pair of matrices could also be created using the *make_matrix* abstraction two times. For example, we could have defined:

```
velocity_u node =                                          (17)
    {    au, aw  =  acceleration node;
         u, w    =  Old_V [ node ] ;
    in   u + au * time };
U  =  make_matrix (grid 50) velocity_u;

velocity_w node =                                          (18)
    {    au, aw  = acceleration node;
         u, w    =  Old_V [ node ] ;
    in   w + aw * time };
W  =  make_matrix (grid 50) velocity_w;
```

However, unlike *make_2_matrices*, the above definitions will unnecessarily compute the *acceleration* twice. In general, the behavior of *make_2_matrices* cannot be imitated by *make_matrix* without loss of efficiency.

In the hydrodynamics application, velocity is indeed represented as a pair of matrices, *U* and *W*, one for each component. However, since the velocity is recomputed in each iteration, both old and new velocities should be pairs of matrices. If *Old_U* and *Old_W* represent the component matrices for the old velocity, then we can define the new velocity matrices as shown below, computing the *acceleration* only once for each component:

```
velocity_uw  node  =                                              (19)
    {   au, aw  =  acceleration  node;
        u       =  Old_U [ node ] ;
        w       =  Old_W [ node ] ;
    in  u + au * time,   w + aw * time };

U, W  =  make_2_matrices  (grid 50)  velocity_uw;                 (20)
```

***Make_matrix_k_ranges:*** The generator *rec_velocity* (definition (15)) which is associated with matrix V of definition (14) can be viewed as a composite of the following two generators:

```
boundary_velocity  node    =  55.5;
interior_velocity  V  node  =  add  (V [ north  node ] , V [ west  node ] );
```

On the north and west boundaries of the matrix, the generator is *boundary_velocity*, and for the remaining matrix the generator is *interior_velocity*. Since the two generating functions are to be applied for "well-defined" patterns of indices *(i,j)*, the evaluation of the conditional for the selection of the appropriate generator function seems to be wasteful. One way to eliminate the condition evaluation is to partition the set of indices into disjoint sets, so that a separate loop with appropriate generator can be used for each set. For example, we can define the following grids:

```
north_boundary  =  ((1, 1), (1, 50));
west_boundary   =  ((2, 50), (1, 1));
interior_nodes  =  ((2, 50), (2, 50));
```

which taken together cover the whole matrix. Now, all we need to do is to "fill" the

matrix using the appropriate generator for each section. For this purpose define a *fill* abstraction as follows:

```
fill_matrix A ((l1, u1), (l2, u2)) f =                      (21)
    {for i from l1 to u1 do
      {for j from l2 to u2 do
        A[i,j] = f (i,j)}};
```

Definition (21) does not return any value and thus is non-functional. However, the single assignment restriction on the structures guarantees the determinacy of the final result. An I-structure behaves like a variable in logic programming in that its value is initially undefined but keeps getting more and more defined as the computation progresses. However, any values "read" from an I-structure always correspond to the final value of the structure. Alternatively, we can think of *A* as in *A* = ***matrix*** *bounds*, as denoting a value which is the set of all matrices with those bounds. Each invocation of *fill_matrix* on matrix *A* is a constraint on the values the variable *A* can assume. Any attempt to fill an element more than once amounts to imposing inconsistent constraints, and hence would result in an error.

Now we can define the matrix abstraction which uses 3 generating functions for 3 ranges of indices:

```
make_matrix_3_ranges dimensions ( (r1, f1), (r2, f2), (r3, f3) ) = (22)
    {   A = matrix dimensions;
        call fill_matrix A r1 f1;
        call fill_matrix A r2 f2;
        call fill_matrix A r3 f3;
    in  A };
```

```
V = make_matrix_3_ranges (grid 50)                                          (23)
  ( (north_boundary, boundary_velocity),
    (west_boundary,  boundary_velocity),
    (interior_nodes, (interior_velocity  V)) );
```

Thus, definition (23) creates the same matrix as definition (15), but avoids the evaluation of the conditional for each element. Once again, all elements will be assigned concurrently. One can generalize the *make_2_matrices* and *make_matrix_3_ranges* and define more general abstractions, such as *make_n_matrices_k_ranges*, for various values of $n$ and $k$. We leave this as an exercise for the reader.

## 3.4. Program for Velocity Computation

We will now illustrate how the velocity computation problem of Section 2 can be coded in *Id*. The velocity is stored in two matrices, $U$ and $W$, one for each velocity component. Since the velocity is computed only for interior nodes, their dimensions should correspond to the interior nodes. The dimensions and the matrix abstraction can be defined as:

$$interior_nodes \;=\; ((kmin,\ kmax),\ (lmin,\ lmax)) \qquad (24)$$

$$new_U, new_W \;=\; make_2_matrices\ \ interior_nodes\ \ velocity; \qquad (25)$$

The velocity is obtained by incrementing the old velocity, $(U,W)$, by the product of time and acceleration. Note that the constant $\delta t$ and the old velocity components $U$ and $W$ will be taken from the context ($\lambda$-*lifted* as discussed earlier).

```
velocity node =                                                    (26)
    {   u_dot, w_dot = acceleration node;
    in  U[node] + δt * u_dot,  W[node] + δt * w_dot};
```

Equation (A) (in Section 2) gives the acceleration at a node. Hence, we define:

```
acceleration node =                                                (27)
    {   d  = nodal_mass node;
        n1 = -(line_integral p z node) - (line_integral q z node);
        n2 =  (line_integral p r node) + (line_integral q r node);
    in  (n1 / d,  n2 / d)};
```

In order to define the line integrals and nodal mass, we need to have the notion of neighbor nodes and zones. Hence we define the following functions that reflect nomen-

clature used in the problem description to refer to the neighboring nodes and zones.

$$
\begin{aligned}
&north\ (i,j) = (i\text{-}1,j); && zone_a\ (k,l) = (k,l);\\
&south\ (i,j) = (i+1,j); && zone_b\ (k,l) = (k+1,l);\\
&east\ (i,j) = (i,j+1); && zone_c\ (k,l) = (k+1,l+1);\\
&west\ (i,j) = (i,j\text{-}1); && zone_d\ (k,l) = (k,l+1);
\end{aligned} \tag{28}
$$

The line integrals described in Section 2 take two quantities, for example, $p$ and $z$. Given a node, they integrate the first quantity with respect to the second along a line around the node. This is expressed by the following line integral function:

$$
\begin{aligned}
&line_integral\ f\ g\ node =\\
&\quad f[\,zone_a\ \ node\,] * (g[\,west\ \ node\,] - g[\,north\ \ node\,]) +\\
&\quad f[\,zone_b\ \ node\,] * (g[\,south\ \ node\,] - g[\,west\ \ node\,]) +\\
&\quad f[\,zone_c\ \ node\,] * (g[\,east\ \ node\,] - g[\,south\ \ node\,]) +\\
&\quad f[\,zone_d\ \ node\,] * (g[\,north\ \ node\,] - g[\,east\ \ node\,]);
\end{aligned} \tag{29}
$$

*Nodal_mass* is the average of the masses of the four zones around a node. This is approximated as one half of the total mass of the 4 zones meeting at node $(k,l)$. The mass of a *zone* is the product of its density and area: $\rho\,[\,node\,] * \alpha\,[\,node\,]$. We can define the following function for mass around a node:

$$
\begin{aligned}
&nodal_mass\ \ node =\\
&\quad 0.5 * (\rho\,[\,zone_a\ \ node\,] * \alpha\,[\,zone_a\ \ node\,] +\\
&\qquad\quad \rho\,[\,zone_b\ \ node\,] * \alpha\,[\,zone_b\ \ node\,] +\\
&\qquad\quad \rho\,[\,zone_c\ \ node\,] * \alpha\,[\,zone_c\ \ node\,] +\\
&\qquad\quad \rho\,[\,zone_d\ \ node\,] * \alpha\,[\,zone_d\ \ node\,]);
\end{aligned} \tag{30}
$$

We believe that definitions (24) through (30) that define the velocity reflect the notions in the problem description of Section 2 very closely. They map the notions of *line_integral*, *acceleration*, neighborhood notions of *nodes* and *zones*, etc., directly into the program. Thus, if changes are to be made to the algorithm, it should be easy to relate to the problem description and change the abstractions as desired.

# 4. Matrix Abstractions in Fortran

The preceding section illustrates how the *Id* style of programming not only helps in building higher-level abstractions, but also automatically unravels the parallelism to the maximum possible extent. Looking at the examples in the preceding section, a casual reader might get the impression that the style is purely syntactic in nature and that such abstractions could be easily built in an imperative language as well. In this section, we will show first, that Fortran lacks certain basic features, such as dynamic storage allocation, which are essential for writing robust abstractions. We will then show that annotations for parallelism in Fortran cannot easily capture the inherent parallelism in the problem specification. Finally, we will comment on the complexity of the annotated programs.

***Basic Abstractions:*** The *make_matrix* abstraction may be expressed as follows in Fortran:

```
      subroutine make_matrix (A, r1, r2, c1, c2, f)
      integer r1, r2, c1, c2
      dimension A (r2, c2)
      do 10 i = r1, r2
      do 10 j = c1, c2
 10   A (i,j) = f(i,j)
      return
      end
```

We would like to show why this program does not capture the subtleties of the *make_matrix* abstraction of (9). First, notice that without the dynamic allocation of storage, we cannot create a new matrix in this subroutine; we can only "fill" a matrix that has been supplied as an argument. Thus, it makes this subroutine the *fill* abstrac-

tion of definition (21) rather than the matrix abstraction of (9). Second, in Fortran, functions are restricted to return one value which must be a scalar. Hence, the above abstraction cannot be used to define a matrix of 2-tuples, such as the velocity matrix of definition (13). In fact, since the type of a matrix in Fortran has to be declared in advance, separate abstractions have to be defined for making integer and floating-point matrices. Restrictions on array bounds further reduces the generality of the abstraction. Note, we have passed *f*, a function, as an argument to this subroutine. The facility to pass function and subroutine names as arguments to other subroutines is not supported in all Fortrans.

The lack of higher-order functions and currying creates a further problem that shows up as an argument-passing problem in Fortran. Suppose we call *make_matrix* with function *g* as the last argument and that *g*, in turn, needs *x* as an argument. In the *Id* version, we would have passed *(g x)*, the curried form of *g*, as an argument to *make_matrix*. However, in Fortran, we will essentially have to write a new version of *make_matrix*, which will take *x* as an additional argument and pass it as an extra argument to *f*. Given all the restrictions on argument passing, it should be easy for the reader to see that there is no way of using the Fortran version of *make_matrix* to define a matrix recursively, as in (14) and (15). Problems only get worse when we try to define *make_2_matrices* in Fortran. One possible candidate for *make_2_matrices* is the following program.

```
      subroutine make_2_matrices (A, B, r1, r2, c1, c2,f)
      dimension A (r2, c2), B (r2, c2)
      do 10 i=r1,r2
      do 10 j=c1,c2
10    call f(i, j, A (i,j), B (i,j))
      return
      end
```

Notice, the argument *f* must be a subroutine with 4 parameters. The first two arguments are the indices *i* and *j*. The last two arguments are values returned and, hence, are bound to the respective elements of the matrix. Such distinction of arguments and return values in a parameter list is inevitable when functions are only allowed to return one scalar value. Given all these problems, the style of programming we have shown in this paper is difficult to copy in the Fortran world. Most of the weaknesses we have pointed out in Fortran so far are not present in sequential languages that support dynamic allocation of storage. A good example of such a language is Common Lisp [18] or its semantically cleaner dialect Scheme [1]. Both of these languages will have some difficulty with the examples involving currying because of "applicative order evaluation".

***Parallelization:*** Now we will examine the parallel execution of Fortran programs. There are many systems that parallelize Fortran. For a recent survey on this subject see [10]. This is usually done by annotating the Fortran program to indicate parallel segments and synchronizations. Different systems use different notations for parallelization of sequential codes. For example, in the subroutine *make_matrix*, the two loops will be executed in parallel by an annotation of the form ***doall*** replacing the "do" in the program. The ***doall*** annotation is specified (by the programmer or by the compiler) after performing subscript analysis of expressions in the loop body.[4] To the best of our knowledge, no parallelization scheme will be able to parallelize the *make_matrix* subroutine because the body invokes some arbitrary function *f*, which can potentially have side-effects that can render the iterations to be dependent and, hence, must be executed sequentially. Complex inter-procedural analysis can sometimes detect that the loops in *make_matrix* may be parallelized when it is invoked with a particular generating

---

4 Although subscript analysis might determine that both of the loops in the nesting can be parallelized, many of the present systems do not have the necessary support for dynamic instantiations and hence restrict that at most one loop can be parallelized within a nesting. Of course, this situation may change in the future.

function *f*. For example, if *f* is the *add* function as in (11) then the loops can be parallelized. But if *f* is the *interior_velocity* function of (23) to create a wavefront matrix, then the iterations cannot be parallelized. Thus, we need different versions of *make_matrix* for different generating functions, which implies that *make_matrix* cannot be a general abstraction for building matrices in Fortran.

In general, it is very hard to extract any parallelism from programs that have some form of recurrence. For example, the wavefront matrix of (23) does have some parallelism along the wavefronts. But in order to make this explicit in Fortran, the programmer must restructure the control substantially. To illustrate this point, consider the following sequential version of an equivalent program. First, the top row and left column are initialized with the constant. Then, the recurrence relation is computed scanning from left to right and top to bottom.

```
      subroutine sequential_wavefront (A, n)
      dimension A (n, n)
      do 10 j = 1, n
      A (1, j) = 55.5
10    A (j, 1) = 55.5
      do 40 i = 2, n
      do 40 j = 2, n
40    A (i, j) = A (i - 1, j) + A (i, j - 1)
```

One way to write a parallel version of wavefront is to traverse diagonally from the top left corner to the bottom right corner. Suppose the diagonals are numbered such that node $(i,j)$ is on diagonal $i+j$. Notice that after elements on diagonal $m$ have been computed, all elements of diagonal $m+1$ can be computed in parallel. Since elements of the top row and left column are initialized to 55.5, we need not traverse diagonals 2 and 3. A complication in coding arises from the fact that the number of elements on a diagonal increases as we traverse from diagonal $4$ to diagonal $n+1$ and decreases as

we traverse from diagonal $n+2$ to diagonal $n+n$. Thus, the loop bounds can be set using conditional statements. The inefficiency of conditional statements can be avoided by splitting the computation into two loops, "DO 45" and "DO 55", (one for each set of diagonals) as shown in the following program. The ***doall*** annotation is used to indicate that all the iterations of a loop should be executed in parallel. The ***barrier*** annotation is used to indicate that all processors must synchronize here before proceeding further. It is worth noting that while ***barrier*** is essential for this program to be correct, it does not permit computations of elements from two diagonals to overlap. This overlapping is possible in the *Id* version shown in definitions (14) and (15).

```
      subroutine wavefront_matrix (A, n)
      dimension A (n, n)
      doall 10 j = 1, n
10    A (1,j) = 55.5
      doall 20 i = 2, n
20    A (i, 1) = 55.5
      barrier
      do 45 m = 4, n + 1
      doall 40 i = 2, m - 2
      j = m - i
40    A (i,j) = A (i - 1,j) + A (i,j - 1)
45    barrier
      do 55 m = n + 2, n + n
      doall 50 i = m - n, n
      j = m - i
50    A (i,j) = A (i - 1,j) + A (i,j - 1)
55    barrier
```

***Complexity due to annotations:*** In the above example, the restructuring and annotations for parallelism obscure the original wavefront algorithm, consequently making it harder to verify its correctness. Any error such as annotating the "DO 45" loop as ***doall*** instead of the "DO 40" loop, or moving the ***barrier*** one line down below the statement 45 will cause non-deterministic results. The error introduced will show up only when iterations are executed in an order that causes old array values to be read instead of the newly updated ones.

As another example to show the advantages of *Id* over Fortran, we develop a parallel version of a program to sum up all the elements of an array. This program is discussed in some detail in [10]. The idea is that given some $p$, we want to divide the array into $p$ bins and compute the partial sums in parallel. Finally, all the partial sums are added up. The following *Id* program accomplishes this (In *Id*, the text between a percentage sign (%) and the end of line is treated as a comment.):

```
cume_segment  A  (l, u)  =
    {    cume  =  0;
    in   { for  i  from  l  to  u  do
              next  cume  =  cume + A [ i ] ;
         finally  cume } };

% n is the size of the array and p is the number of processors
p_way_accumulate_array  A  n  p  =
    {    bin_size  =  fix  ((n + p - 1) / p);     % fix gives the ceiling of n/p

         % define a function to the compute index bounds for bin j
         bin_bounds  j  =  (j - 1) * bin_size + 1,   (min  n  (j * bin_size));

         % define a function to accumulate elements of bin j
         cume_bin  j  =  cume_segment  A  (bin_bounds  j);

         B = make_array  (1, p)  cume_bin;
    in  cume_segment  B  (1, p) } ;
```

We invite the reader to compare this program with the annotated Fortran version in [10] for clarity, parallelism and efficiency. It should be noted that in the *cume_segment* function shown above, although the array elements may be accessed concurrently, the summation is done sequentially as dictated by the recurrence relation *cume*. It is possible to incorporate abstractions that make use of the commutative and associative properties of a function to permit its applications in arbitrary order. In fact, an experimental version of such an abstraction is available in *Id*.

# 5. Optimizing the Velocity Program

Programming at a high level involves defining many functions, sometimes very simple functions such as *north, south, etc.* as seen in the velocity example. In any implementation, each function call introduces computational and other resource overheads. Programs targeted for maximum efficiency tend to eliminate such overheads by encoding the program at such a low level that the program often does not reflect the problem at all. This point is illustrated by the following equivalent Fortran program taken from a running version of Simple.

```
C    COMPUTE ACCELERATION AND NEW VELOCITIES
   DO 100 L=LMN,LMX
    DO 110 K=KMN,KMX
     AU= (P(K,L)+Q(K,L))      *(Z(K,L-1)-Z(K-1,L)) +
 1     (P(K+1,L)+Q(K+1,L))  *(Z(K+1,L)-Z(K,L-1)) +
 2     (P(K,L+1)+Q(K,L+1))  *(Z(K-1,L)-Z(K,L+1)) +
 3     (P(K+1,L+1)+Q(K+1,L+1))*(Z(K,L+1)-Z(K+1,L))
     AW= (P(K,L)+Q(K,L))      * (R(K,L-1)-R(K-1,L)) +
 1     (P(K+1,L)+Q(K+1,L))  * (R(K+1,L)-R(K,L-1)) +
 2     (P(K,L+1)+Q(K,L+1))  * (R(K-1,L)-R(K,L+1)) +
 3     (P(K+1,L+1)+Q(K+1,L+1))* (R(K,L+1)-R(K+1,L))
     AUW = RHO(K,L)*AJ(K,L)   +RHO(K+1,L)*AJ(K+1,L)+
 1      RHO(K,L+1)*AJ(K,L+1)+RHO(K+1,L+1)*AJ(K+1,L+1)
     AUW = 2./AUW
     AU = -AU*AUW
     AW = AW*AUW
     U(K,L) = U(K,L)+DTN*AU
     W(K,L) = W(K,L)+DTN*AW
 110  CONTINUE
 100 CONTINUE
```

A good Fortran compiler can analyze all the expressions in the above program and deduce that all the iterations in the two nested loops can be executed concurrently. The analysis has to be fairly sophisticated, as it must deduce that *P, Q, R, Z, RHO, AJ* are all constant matrices (as far as the loop is concerned) and that the elements of *U* and *W* are computed pointwise from their old values without forming any recurrences. In general, this may require checking that any side-effects caused by functions invoked within these subcomputations are non-interfering, so that they could be performed in arbitrary order. Any slight change to the program affecting the above decisions could make the compiler decision much harder. Alternatively, the programmer (who figures out all these facts) can annotate the program by specifying ***doall*** in place of ***do*** in the two loops. To a large extent, the programmer had already done a number of optimizations, such as the use of actual indices (as opposed to functions like ***north***), and grouping and factoring of expressions in order to minimize the number of operations, memory references, etc. In addition, Fortran compilers usually produce very efficient code for programs like this because the programs are highly "expression-oriented" and, therefore, are amenable to optimizations such as common subexpression elimination, strength reduction, moving out constant expressions from loop bodies, etc. It is extremely hard to decipher the algorithm used in the above program. Consequently, it is also hard to modify it for minor changes in the algorithm. One can imagine the added complexity if this task is to be done with multiple processors in mind.

We have already shown that we can code this algorithm in *Id* without obscuring its structure. The reader can verify, by applying the rewrite rules given in Section 3.1, that the inherent parallelism of the algorithm will be exposed during execution. Finally, we show that the efficiency of the Id program matches that of the Fortran program, when in-line substitutions for functions are made. Starting from (25) we performed in-line substitutions for all the functions (24) through (30), and then performed a simple automatic renaming transformation to eliminate nested block definitions. Finally, standard common subexpression elimination transformations were performed to get the following program:

```
new_U, new_W =                                                    (31)
    {A = matrix ((kmin, kmax), (lmin, lmax));
    B = matrix ((kmin, kmax), (lmin, lmax));
    {for i from kmin to kmax do
      {for j from lmin to lmax do
        d = 0.5 * (ρ [ i,j ] * α [ i,j ] +
          ρ [ i+1,j ] * α [ i+1,j ] +
          ρ [ i+1,j+1 ] * α [ i+1,j+1 ] +
          ρ [ i,j+1 ] * α [ i,j+1 ] );
        z1 = z [ i,j-1 ] - z [ i-1,j ] ;
        z2 = z [ i+1,j ] - z [ i,j-1 ] ;
        z3 = z [ i,j+1 ] - z [ i+1,j ] ;
        z4 = z [ i-1,j ] - z [ i,j+1 ] ;
        r1 = r [ i,j-1 ] - r [ i-1,j ] ;
        r2 = r [ i+1,j ] - r [ i,j-1 ] ;
        r3 = r [ i,j+1 ] - r [ i+1,j ] ;
        r4 = r [ i-1,j ] - r [ i,j+1 ] ;
        pq1 = p [ i,j ] + q [ i,j ] ;
        pq2 = p [ i+1,j ] + q [ i+1,j ] ;
        pq3 = p [ i+1,j+1 ] + q [ i+1,j+1 ] ;
        pq4 = p [ i,j+1 ] + q [ i,j+1 ] ;
        n1 = - (pq1 * z1 + pq2 * z2 + pq3 * z3 + pq4 * z4);
        n2 = pq1 * r1 + pq2 * r2 + pq3 * r3 + pq4 * r4;
        u_dot, w_dot = n1 / d,  n2 / d;
        A [ i,j ],  B [ i,j ] = U [ i,j ] + δt * u_dot,  W [ i,j ] + δt * w_dot
    in A, B}
```

The reader can verify that the above program does the same number of *arithmetic* operations, *loads* and *stores* as the Fortran version given earlier.

# 6. Storage Requirements

A common criticism of purely functional programs is that they require too much storage. Indeed, this was the main reason for the introduction of operators to update a storage cell in McCarthy's functional Lisp [12] soon after it was invented. Functional languages, traditionally, have either paid no attention to arrays or have simulated them using lists. When scientific programs are written in such languages, excessive copying of data structures and poor utilization of storage should not surprise anybody. Furthermore, random accessing of array elements can also cause great inefficiencies. The situation in *Id* is different because of I-structures. First of all, the storage for a matrix can be allocated contiguously whenever ***matrix*** *((l1,u1),(l2,u2))* is executed. Thus, any element of the matrix can be selected in constant time. Second, there is no unnecessary copying of data structure elements when *all* elements for a data structure are defined anew, as in relaxation algorithms. This point is rather subtle and is often not understood by novice *Id* programmers who may be experienced Fortran programmers. *Id* provides even greater flexibility than other functional languages like Val and Sisal which do support arrays. In Val and Sisal [13,14] arrays are "strict", that is, no element of an array may be read until the whole array has been defined. Consequently, programs like wavefront (see equations (14) and (15)) cannot be expressed efficiently.

It should be emphasized that the use of additional storage in functional model is caused mostly to facilitate parallel operations. In the Fortran environment, often a programmer designs the memory structures and program control in such a manner that the same memory area can be reused by many subcomputations. But, invariably, this is based on restricting the evaluation order to be sequential. When subcomputations can potentially execute concurrently, such memory sharing is neither desirable nor possible. When Fortran programs are parallelized, additional storage in terms of private copies of variables is indeed provided to run subcomputations in parallel. We will

illustrate this more concretely through the following example of LU decomposition, whose parallelization is discussed extensively in [10].

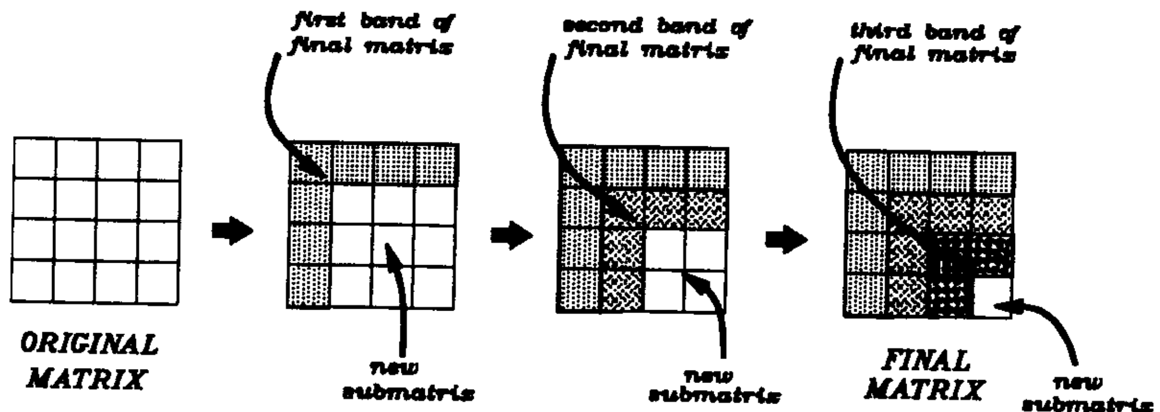



Figure 4: Computation of successive bands of the final matrix yielding LU decomposition

The goal of LU decomposition is to decompose an $n \times n$ matrix into lower and upper triangular matrices by Gaussian elimination method. In the Fortran program discussed in [10], the decomposed matrix is produced by successively modifying it, as shown in Figure 4. In each iteration, a band of the final matrix is produced and the remaining submatrix is completely modified. Thus, iteration $k$ modifies only the right submatrix bounded by the *k-th* row and *k-th* column. The steps executed in each iteration are shown in Figure 5. First, the pivot is determined by finding the maximum element in the first column of the submatrix. Then, the pivot row and top row are interchanged. Next, the elements of the first column (other than the pivot) are multiplied by a multiplier. The top row and left column of the submatrix created in this manner forms the next band of the final matrix. Finally, the new submatrix is computed by incrementing each element with the product of the corresponding elements in the band created above.

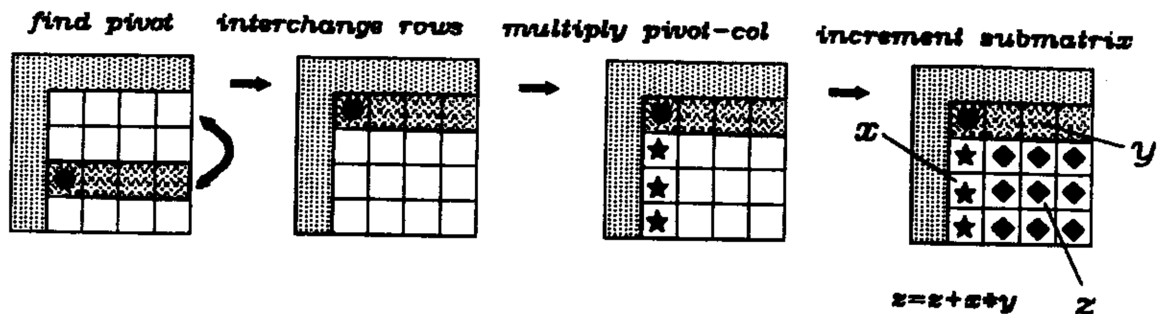



Figure 5: Steps in pivoting each diagonal element

Since functional languages do not permit modification of the matrix, we must produce the LU decomposition in a separate matrix. However, in this example, it is not necessary to copy the portions of the matrix each time. Initially, we allocate the matrix $D$ which will contain the final LU decomposition. In each iteration, we fill in a band of $D$. Instead of modifying the submatrix in each iteration, we create a new submatrix each time. The submatrices in successive iterations have smaller and smaller dimensions. Similarly, instead of interchanging the rows of a submatrix, we will select the elements of the appropriate row. The array $P$ gives the indices which keep track of the order of interchanges needed for pivoting. The following *Id* program performs these operations.

```
% finds the largest element in first column and returns that row number
find_pivot  A =                                                          (32)
    {   ((l1, u1), (l2, u2))  =  bounds  A;
        m  =  l1;
    in  { for  i  from  l1 + 1  to  u1  do
              next  m  =  if  A [ i, l2 ]  >  A [ m, l2 ]  then  i  else  m;
        finally  m } };
```

```
decompose A n =                                                        (33)
  { D = matrix ((1,n),(1,n));
    P = array (1,n);
    B = { for k from 1 to n-1 do
          r = find_pivot A;  % get number of pivot row
          P[k] = r;   % record the row interchange

          % fill the row of the new band with elements from pivot-row
          call fill_matrix D ((k,k),(k,n)) row_fill;
          row_fill (i,j) = A[r,j] ;

          % multiply pivot-col elements with multiplier t
          call fill_matrix D ((k+1,n),(k,k)) col_fill;
          col_fill (i,j) = A[i,k] * t ;
          t = -1.0 / A[r,k] ;  % compute multiplier

          % compute elements of new submatrix
          % new-element = old-element + product of corresponding band elements
          % Because of interchange, pivot-row should be interpreted as the top row
          [illegible] if i==r then k else i
          new_element (i,j) = D[i,k] * D[k,j] + A[[illegible], j] ;
          next A = make_matrix ((k+1,n),(k+1,n)) new_element };
      finally A };
    D[n,n] = B[n,n] ;
  in D, P };
```

In the parallel Fortran versions of this program [10], the elements of the submatrix are computed in parallel and are stored back in the same storage area of the matrix $A$.

However, in order to guarantee determinacy, barrier synchronization must be used to prevent the next iteration to commence before all the elements of the submatrix are computed. In (33) the submatrices are created in separate memory areas, but the iterations can overlap. For instance, finding the pivot for iteration $k+1$ can commence as soon as the first column of the submatrix ***next*** $A$ is computed in iteration $k$. Thus, the *Id* program uses additional storage to expose all possible parallelism. It is possible to constrain the execution of the loop in (33) so that the iterations execute sequentially. Then, the storage for the matrix $A$ can be reclaimed as soon as the iteration is completed. Thus, in functional programs also, it is possible to save storage at the expense of constraining some parallelism. Similarly, when subcomputations are known to run sequentially, one can use clever memory management schemes which can take directives so that copying can be avoided in certain cases. The *Id World* environment [16], provides facilities to experiment with such trade-offs between use of additional memory and exploiting parallelism. In general, management of memory and processor resources in a parallel environment is an open problem and a subject for further study [7].

In spite of all the flexibility offered by I-structures, there are well known graph algorithms that cannot be expressed efficiently in *Id*. Warshall's algorithm for computing the transitive closure causes unnecessary copying of matrix elements. We are keen to investigate if this points to a lack of expressive power in *Id* or if it is a strong indicator that such algorithms do not have parallelism.

# 7. Conclusion

Ideally, a high-level language should provide a way of writing abstractions which are as close to the problem domain as possible. It should also facilitate efficient implementations of these abstractions lest a user try to "get underneath" the abstractions. With the advent of parallel machines, a language such as Fortran fails on both counts. It was never very good for expressing high-level abstractions and, because it forces the user to specify a sequential order of evaluation, it also makes it very difficult to compile good code for a parallel machine. In the latter deficiency, Fortran is not alone; all high-level languages in widespread use today force the user to over-specify the algorithm. *Functional* and other *declarative* languages offer relief on both counts. In this paper, using the *make_array* and *make_matrix* abstractions, we have shown that the use of higher-order functions in *Id* raises the level of programming without loss of parallelism or efficiency.

We have also discussed our attempts to write the *make_matrix* and other abstractions in Fortran. The primary difficulty in expressing these abstractions in Fortran has to do with the lack of dynamic storage allocation. In Fortran, one cannot allocate an array in a function or subroutine and return it as a result. This restriction allows a user (with the help of the Fortran compiler) to make very efficient use of storage but often results in quite inelegant programs. The other major shortcoming of Fortran and most other imperative languages is the lack of support for curried higher-order functions. Without currying, there is no elegant way of dealing with functions with a variable number of arguments.

We would like to caution against hasty reactions like: "*Since there are only two deficiencies, - dynamic storage allocation and currying - let us add them to Fortran.*" Or "*Since this other imperative language has all these features, why not use it?*" It is of

course necessary to have these features for specifying high-level abstractions. But it is also important to design the language such that its operational semantics does not impose unnecessary sequentiality. Parallelism in the operational semantics is ensured by the *functional* nature of a language. Although freewheeling use of I-structures can take us beyond Functional programs, *Id* has been designed to retain the Church-Rosser property and all the inherent parallelism of functional programs. Unregulated mixing of imperative and functional features in a language essentially makes the language imperative, because it is extremely difficult to determine if a construct is functional or not. Again Lisp offers a good example to illustrate this point; pure Lisp did not specify the order in which arguments of a function should be evaluated. However, introduction of just two imperative features to destroy the contents of a cons-cell mandate sequential order of evaluation. We think, without changing an imperative language in a fundamental way, it is impossible to make it suitable for abstract programming and parallel execution.

Annotations in Fortran for parallel execution make a bad situation much worse. Incorporation of parallel-loop constructs and synchronization primitives permits the user to write programs whose behavior may inadvertently be time-dependent or configuration-dependent. This adds a new and treacherous dimension to debugging programs. In contrast, programs in *declarative* languages are determinate. Such languages eliminate the problem of "detecting parallelism"; however, the problem of managing resources for parallel execution remains. In our opinion it is still not fully appreciated that parallel execution of Fortran programs requires at least a limited notion of dynamic storage allocation, and invariably takes more storage than sequential execution. It may be best to adopt new and progressive ways at this stage rather than beat a three-legged horse!

***Acknowledgements:*** We wish to thank Olaf Lubeck and Rishiyur Nikhil for their helpful comments on earlier drafts of this paper. We thank John Single, Richard Soley and Natalie Tarbet for their editorial changes. Finally without Ken Traub's *Id* compiler, this study of Simple code would not have been possible. We are specially thankful to him for his prompt and helpful responses on compiler-related issues. Funding for the Massachusetts Institute of Technology Laboratory for Computer Science is provided in part by the Advanced Research Project Agency of the U.S.Department of Defense under the office of Naval Research contract N00014-84-K-0099.

# 8. Bibliography

[1] Abelson, H. and G.J.Sussman, "Structure and Interpretation of Computer Programs", The MIT Press, Cambridge, MA (1985).

[2] Arvind and R.S.Nikhil, "Executing a Program on the MIT Tagged-token Dataflow Architecture", Proceedings of PARLE conference, Eindhoven, The Netherlands. Springer-verlag LNCS 259 (June 1987).

[3] Arvind, R.S.Nikhil, K.K.Pingali, "I-structures: Data Structures for Parallel Computing", Proceedings of Workshop on Graph Reduction, Santa Fe, NM. Springer-verlag LNCS 279 (September 1986).

[4] Arvind, R.S.Nikhil and K.K.Pingali, "Id Nouveau Reference Manual, Part II: Operational Semantics", CSG Memo, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA (April 1987).

[5] Backus, J., "Can Programming be liberated from the von Neumann style? A Functional Style and Algebra of Programs", Communications of the ACM, Vol 21, No 8 (August 1978).

[6] Crowley, W.P., C.P.Hendrickson and T.E.Rudy, "The SIMPLE Code", Technical Report UCID 17715, Lawrence Livermore National Laboratory, University of California, Livermore, CA (February 1978).

[7] Culler, D.E., "Effective Dataflow Execution of Scientific Applications", Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA (expected December 1988).

[8] Ekanadham, K. and Arvind, "SIMPLE: Part I - An Exercise in Future Scientific Programming", Technical Report RC 12686, IBM T.J.Watson Research Center, Hawthorne, NY (April 1987).

[9] Johnsson, T., "Lambda Lifting: Transforming Programs to Recursive Equations", Proceedings of Functional Programming Languages and Computer Architecture, Nancy, France. Springer Verlag LNCS 201 (September 1985).

[10] Karp, A.H., "Programming for Parallelism", Computer, Vol 20, No 5, pp.43-57 (May 1987).

[11] Lubeck, O., J.Moore and R.Mendez, "A Benchmark Comparison of 3 Supercomputers: Fujitsu VP-200, Hitachi S810/20 and Cray X-MP/2", Computer, Vol 18, No 12, pp.10-24 (December 1985).

[12] McCarthy, J., "Recursive Functions of Symbolic Expressions and their Computation by Machine", Communications of the ACM, Vol 3, No 4, pp.184-195 (April 1960).

[13] McGraw, J.R., "The VAL Language: Description and Analysis", ACM Transactions on Programming Languages and Systems, Vol 4, No 1, pp.44-82 (January 1982).

[14] McGraw, J.R. *et al*, "SISAL: Streams and Iteration in a Single Assignment Language" - Language Reference Manual, version 1.2, Technical Report M-146, Lawrence Livermore National Laboratory, University of California, Davis, CA (March 1985).

[15] Nikhil, R.S., "Id Nouveau Reference Manual, Part I: Syntax", CSG Memo, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA (April 1987).

[16] Nikhil, R.S., "Id World Reference Manual", CSG Memo, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA (April 1987).

[17] Padua, D.A. and M.J.Wolfe, "Advanced Compiler Optimizations for Supercomputers", Communications of the ACM Vol 29, No 12, pp.1184-1201 (December 1986).

[18] Steele, G.L., "Common Lisp: The Language", Digital Press, 30 North Ave., Burlington, MA (1984).

[19] Traub, K.R., "A Compiler for the MIT Tagged-token Dataflow Architecture", Technical Report TR-370, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA (August 1986).

[20] Turner, D.A., "The Semantic Elegance of Applicative Languages", Proceedings of ACM Conference on Functional Programming Languages and Computer Architecture, Portsmouth, NH (October 1981).