CONF-9006303--1

UCRL-JC-106081
PREPRINT

Arrays in Sisal

John Feo
Lawrence Livermore National Laboratory
Livermore, CA

This paper was prepared for submittal to
the First International Workshop on Arrays,
Functional Languages, and Parallel Systems
Montreal, Canada
June 12-14, 1990

September 1990

UCRL-JC--106081

DE91 005400

Arrays in Sisal

John T. Feo

Computer Research Group (L-306), Lawrence Livermore Nat. Lab., P.O. Box 808, Livermore, CA 94550

Although Sisal (Streams and Iterations in a Single Assignment Language) is a general-purpose applicative language, its expected program domain is large-scale scientific applications. Since arrays are an indispensable data structure for such applications, the designers of Sisal included arrays and a robust set of array operations in the language definition. In this paper, we review and evaluate those design decisions in light of the first Sisal compilers and runtime systems for shared-memory multiprocessor systems. In general, array intensive applications written in Sisal 1.2 execute as fast as their Fortran equivalents. However, a number of design decisions have hurt expressiveness and performance. We discuss these flaws and describe how the new language definition (Sisal 2.0) corrects them.

Introduction

Sisal 1.0 [7] was defined in 1983 and revised in 1985 (Sisal 1.2 [8]). The language definition was a collaborative effort by Lawrence Livermore National Laboratory, Colorado State University, University of Manchester, and Digital Equipment Corporation. Although Sisal is a general-purpose applicative language, its expected program domain is large-scale scientific applications. Since arrays are an indispensable data structure for such applications, the designers of Sisal included arrays and a robust set of array operations in the language definition.

Unfortunately, defining and implementing arrays under applicative semantics is not easy. There are three major problems. First, an array, like any object, is the result of an expression. The idea of allocating storage and filling in values is alien to applicative semantics. Second, not all array definitions are legal. Since an array element can be defined at most once, any recursive definition which defines some elements more than once is illegal. Third, operations which modify arrays must first copy their operands. The cost of copying large array is prohibitive.

To solve the first problem, designers of applicative and higher-order functional languages have developed *array comprehensions* and *gather clauses*. Array comprehensions, or monolithic arrays, permit the definition of subregions of an array within a single expression. For example, a 4 x 4 block diagonal unit matrix is defined in Sisal 2.0 as

```
X := array [1..4, 1..4:
        [1..2, 1..2] 1, 1, 1, 1;
        [1..2, 3..4] 0, 0, 0, 0;
        [3..4, 1..2] 0, 0, 0, 0;
        [3..4, 3..4] 1, 1, 1, 1]
```

Sisal 2.0 includes array comprehensions, but Sisal 1.2 does not. In Sisal 1.2, arrays are built primarily by loop expressions that deterministically assemble loop values into arrays via gather clauses. For example, vector sum is defined in Sisal 1.2 as

```
X := for i in 1, n
        z := A[i] + B[i]
     returns array of z
     end for
```

The loop body generates n z-values which are formed into an array by the returns clause.

What to do about recursive array definition is a difficult problem with no good solution. One possibility is to exclude recursive definitions all together; however, this restricts a language's ability to express certain computations and may obscure parallelism. A second alternative is to accept only those recursive definitions that can be proved legal at compile time. Recent work in this area appears promising [4]. Finally, a language designer may decide to accept all recursive definitions and rely on hardware to trap illegal expressions. This solution maximizes expressiveness, but requires special hardware and without a fine-grain implementation may introduce deadlock. Recent studies have shown fine-grain implementations to be impractical [2].

For applicative programs to execute as fast as imperative programs, copy elimination is essential. We cannot overcome the cost of copying large arrays by increased parallelism – there is simply not enough parallelism. Copy elimination involves three levels of analysis: static inferencing, node reordering, and reference counting. Compile-time analysis can often identify the last user of an array. Reordering the graph to schedule read operations before write operations improves the chances of in place operations. In those instances in which analysis fails, the compiler can insert reference count operations and runtime checks to identify the last user at runtime. But if done improperly, reference counting can become a bottleneck and degrade performance [11]. In recent years researchers have made tremendous progress in the general area of copy elimination and reference count optimization in applicative and higher-order functional languages [1,5,14]. The work by Cann [1] has virtually eliminated the copy problem in Sisal.

Although conservative, the Sisal 1.2 approach has been successful. The language includes gather clauses, but neither array comprehensions nor recursive definitions (in fact, the compiler enforces a strict "definition before use" style). To eliminate copy operations, the native code compiler rearranges nodes, introduces artificial dependencies to schedule readers before writers, and inserts runtime checks when analysis fails [1]. We find that we can express most array computations easily and concisely in Sisal 1.2, and most array intensive applications execute as fast as their Fortran equivalents on shared-memory multiprocessors [3].

This paper is organized as follows. Section two describes arrays and array operations in Sisal 1.2 and discusses changes to arrays in the new language definition, Sisal 2.0 [12]. Section three presents an efficient solution for Gaussian elimination using gather operations. We compare the Sisal 1.2 code to an equivalent Fortran program on the Alliant FX/80. Section 4 discusses an array computation which is difficult to express in Sisal 1.2, but easy to express in Sisal 2.0. In section 5, we conclude by describing some of the new language directions we are persuing.

Arrays in Sisal 1.2

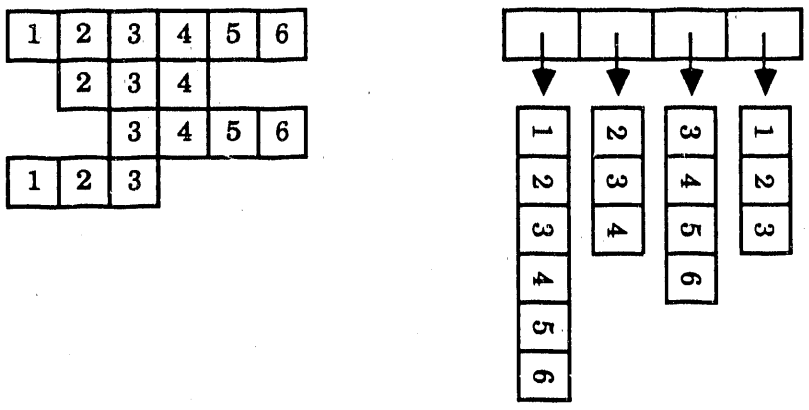
Array Declaration

Sisal 1.2 includes the standard scalar types: integer, real, double precision, boolean, and character. All other types are user-defined. An array declaration specifies only the component type (any scalar or user-defined type). It does not specify size, lower or upper bound, or structure – these are specified by the expressions that create arrays. An array of integers is defined as

```
type OneDim = array[integer]
```

In Sisal 1.2, an *n*-dimensional array is an array of *(n-1)*-dimensional arrays. A two-dimensional array of integers is defined as an array of array of integers,

```
type TwoDim = array[OneDim]
```



a) Logical b) Storage

Figure 1 - Hierarchial Ragged Arrays

The components of `TwoDim` can be either the rows or the columns of the mathematical array. Since the elements of an n-dimensional array are arrays, the size and bounds of each element may be different. Thus, Sisal 1.2 arrays are *hierarchical* and *ragged* [Figure 1a].

Hierarchical arrays are convenient when expressing row- or column-ordered algorithms, permit row sharing, and reduce copying costs. However, they make read operations more expensive, prevent easy access to arbitrary subcomponents (such as blocks or diagonals), and make expressing anything but a row- or column-ordered algorithm difficult. Raggedness is ideal when programming an array whose rows (columns) have different or continually changing sizes and bounds. In hydrodynamics codes, an array is typically used to represent a grid of cells – each cell comprising an arbitrary number of particles. During execution, particles move from cell to cell. The continual change in cell size is easily programmed using ragged arrays. In LU-decomposition, the lower and upper triangular matrices may be stored as two ragged arrays in minimum space. However, ragged arrays do not support strides, make vectorization difficult, and require a hierarchical storage implementation [Figure 1b].

For the great majority of applications which do not need hierarchical ragged arrays, the drawbacks are severe. But, since the advantage to some applications is great, Sisal 2.0 includes both hierarchical ragged arrays and flat arrays.

Array Creation

Any expression which has type *array* creates a new array. The simplest way to create an array in Sisal 1.2 is to list the elements,

```
r := array OneDim []
s := array OneDim [1: 1,2,3]
t := array TwoDim [1: array [1: 1,2,3], array [1:4,5,6]]
```

The first expression creates an empty array of integers. The type specification is mandatory. The second expression builds an array of integers with lower bound 1. The type specification is optional since it can be derived from the element list. The third expression builds a two-dimensional array (an array of arrays) of integers. *t* is similar to a flat array with two rows and three columns.

The most common way to create arrays in Sisal 1.2 is by loop expressions. The **for** expression provides a means to specify independent iterations. This expression's semantics does not allow references to values defined in other iterations. Recursive definitions are not permitted. A **for** expression comprises three parts: a range generator, a loop body, and a returns clause. Consider the expression

```
u := for i in 1,n cross j in 1,m
       z := i + j
     returns array of z
     end for
```

An instance of the loop body is executed for each *(i, j)* pair, $1 \leq i \leq n, 1 \leq j \leq m$. The returns clause, **array of**, deterministically gathers the *z* values into an array. The order of reduction, size, and structure of the array are specified by the range generator. Thus, *u* is a two-dimensional array with *n* rows, *m* columns, and $u[i, j] = i + j$.

For expressions are expressive, flexible, easy to implement, and exhibit good speedup on medium- and coarse-grain shared memory multiprocessors. Because the loop bodies are independent and recursive definitions are not permitted, the runtime system may execute the subexpressions in any order. Compile-time analysis inserts code to preallocate array storage where analysis or runtime calculations can determine array sizes [13]. Thus, most results are built in place eliminating useless copying [3]. **For** expressions do have one flaw – they couple the scattering of work and the gathering of result values. This is unnecessary, and results in confusion and programming errors. A common mistake is to write the transpose of an *(n x m)* matrix as

```
for i in 1,n cross j in 1,m
returns array of X[j, i]
end for
```

This expression returns an *(n x m)* matrix, not an *(m x n)* matrix. The correct expression is

```
for i in 1,m cross j in 1,n
returns array of X[j, i]
end for
```

In Sisal 2.0, the scattering of work and gathering of results are decoupled. The former governed by the range generator and the latter governed by the returns clause.

For initial expressions resemble sequential iteration in conventional languages, but retain single assignment semantics. They comprise four parts: initialization, loop body, termination test, and returns clause. The initialization segment defines all loop constants and assigns initial values to all loop-carried names. It is the first iteration of the loop. The loop body computes new values for the loop names. Unlike **for** expressions, loop values on the previous iteration are accessible – ***old*** *<name>* returns the value of *<name>* on the previous iteration. The returns clause, **array of**, gathers loop values in order into an array.

Although Sisal 1.2 prohibits recursive definitions, **for initial** expressions can build recursively defined arrays. Consider the array definition

$$X(i,j) = \begin{cases} 1 & i = 1 \\ 1 & j = 1 \\ X(i, j-1) + X(i-1, j) & 2 \le i \le n,\ 2 \le j \le n \end{cases}$$

A legal Sisal 1.2 expression for *X* is

```
X := for initial
       i   := 1;
       row := array_fill(1, n, 1);
     while i < n repeat
       i   := old i + 1;
        row := for initial
                 j := 1;
                 x := 1
```

```
                while j < n repeat
                  j := old j + 1;
                  x := old x + old row[j]
                returns array of x
                end for
        returns array of row
        end for
```

The expression is certainly long and obscures parallelism. It is hard to ascertain that computations along a diagonal are data independent. However, a nonstrict implementation of the loop bodies – initiate all loop bodies simultaneously and have each wait for its inputs – will realize the parallelism.

If **for initial** expressions can build recursively defined arrays and do not necessarily limit parallelism, has excluding recursive definitions cost us anything? Definitely, yes! A **for initial** expression can compute new loop values as a function of only the loop values of the previous iteration. Worse yet, **returns array of** binds the ith element of the result to the ith iteration. These two restrictions greatly complicate the programming effort and may increase copying costs. Sisal 2.0 still prohibits recursive array definitions (the performance consequences are just too great), but does allow users to specify where values computed on the *ith* iteration appear in the results.

Other Array Operations

Read operations have the form,

```
        x := A[1]
```

The type of *x* is the component type of *A*. The shorthand notation, *A[i, j]*, returns the *jth* component of the *ith* component of *A*. In Sisal 1.2, you can read (access) only single elements. This is unfortunate because many array computations operate along diagonals or within blocks of an array. Sisal 2.0 permits reads to any contiguous set of elements separated by a constant stride.

Write operations have the form,

```
        x := A[1: 0, 1, 2]
```

The expression preserves single assignment semantics by creating a new array, *x*, which is identical to *A* except that the first, second, and third elements are *0*, *1*, and *2*, respectively. The inclusion of such explicit write operations simplifies copy elimination analysis. If the compiler can ascertain, or a runtime check can assure, that the write is the last consumer of *A*, then the update can execute in place. Although write operations can change several element values at once, the syntax is still too limiting. Values along diagonals or within blocks cannot be changed in a single expression. Like reads, Sisal 2.0 permits writes to any contiguous set of elements separated by constant stride.

Sisal 1.2 also includes operations to concatenate arrays, insert or remove values at either end of an array, set the bounds of an array, and return an array's size. Except for concatenation and array size, the more general syntax of read, write, and array creation in Sisal 2.0 supplants the other operations. It remains to be seen whether we have overly complicated copy elimination analysis by replacing explicit array modification operations with more general expressions.

LU Decomposition

LU decomposition is a method to solve systems of linear equations of the form

$$A\ x = b$$

where A is an $n \times n$ matrix, and x and b are $n \times 1$ column vectors. The method reduces A into a lower *(L)* and upper *(U)* triangular matrix. A common solution method for LU decomposition is Gaussian elimination without pivoting. The algorithm comprises n iterative steps. At step i, rows $i + 1$ to n are reduced by row i. Row i is called the pivot row and $A[i, i]$ is called the pivot element. The reduction executes in two steps:

1. $A[k, i] = A[k, i] \,/\, A[i, i], \qquad i+1 \le k \le n$

2. $A[k, l] = A[k, l] - A[k, i] * A[i, l], \quad i+1 \le k, l \le n$

L comprises the n column vectors computed in step 1 (the multipliers) and U comprises the n pivot rows.

Since A is modified every iteration, a functional implementation of Gaussian elimination creates $n - 1$ intermediate copies of A. As the computation progresses, the number of computations decreases and the amount of computationless copying increases. Solutions have ranged from proposing new algorithms [6,10] to extending the idea of monolithic arrays [4]. However, Sisal's gather operations solve the problem naturally and efficiently.

Consider the following functions

```
type OneDim = array[double_real]
type TwoDim = array[array[double_real]]

function GE(n: integer; A_in: TwoDim returns TwoDim, TwoDim)
  for initial
    i := 1;
    P := A_in[1];
    M := Multipliers(i, n, A_in);
    A := Reduce(i, n, M, A_in)
  while i < n repeat
    i := old i + 1;
    P := old A[i];
    M := Multipliers(i, n, old A);
    A := Reduce(i, n, M, old A)
  returns array of M
          array of P
  end for
end function % Gaussian Elimination

function Multipliers(i,n:integer; A: TwoDim returns OneDim)
  for k in i+1, n
  returns array of A[k,i] / A[i,i]
  end for
end function % Multipliers

function Reduce(i,n: integer; M: OneDim; A: TwoDim
                returns TwoDim)
  for k in i+1, n cross l in i+1, n
  returns array of A[k,l] - M[k] * A[i,l]
  end for
end function % Reduce
```

In **function** GE, *P* is the pivot row, *M* is the vector of multipliers, and *A* is the reduced matrix. Notice the number of rows and columns in *A* decrease by one every iteration. There is no copying and no useless computation. Every computation is necessary and computes a new value. The vectors of multipliers and pivot rows are gathered to form *L* and *U*. Both arrays are ragged; however, it is easy to extend the code to return rectangular arrays. Because the gather operation removes *P* and *M* from the computation, the code avoids useless copying. The completed portion of *L* and *U* are not carried (i.e., copied) from iteration to iteration. Unlike the equivalent Fortran code which stores the reduced matrix back into *A*, the Sisal code preallocates new storage for *A* and deallocates the storage for ***old** A* every iteration. Although this is cheap, it is not an insignificant expense.

We compared equivalent Sisal and Fortran versions of Gaussian elimination on a 200 x 200 problem on the Alliant FX/80. The Sisal execution times on one and five processors were 7.53 and 2.30 seconds, respectively. The Fortran execution times were 7.09 and 1.42 seconds, respectively. Memory management operations cost Sisal 1.15 and 0.61 seconds on one and five processors, respectively. In all, Sisal was 6% slower than Fortran on one processor and 62% slower on five processors. Because memory deallocation is sequential, the speedup of the Sisal code was only 3.3.

Clearly, not reusing storage is hurting performance. There are two solutions. One possible optimization is to allocate space for two $n \times n$ arrays outside the **for initial** expression and use the two arrays alternatively for A and ***old** A*. At the end of each iteration we would swap pointers instead of allocating and deallocating memory. A second solution is to use flat arrays. We have found repeatedly that managing hierarchical arrays is expensive. The cost of allocating and deallocating a hierarchical array is linear in the product of the sizes of the outer dimensions, and the cost of accessing the innermost component is linear in the number of dimensions. For flat arrays the cost of both these operations is constant. We expect the Sisal 2.0 version of Gaussian elimination, which will include both optimizations, to execute as fast as the Fortran code.

Segmental Recomputation

Segmental recomputation is a simplification of the second Livermore Loop [9]. Let X and V be two n element vectors (assume n is a power of 2). Compute new values for X as follows:

1. Initialize *LHS, RHS,* and *VHS* as:

$$X[1 .. \frac{n}{2}], \quad X[\frac{n}{2}+1 .. \frac{3n}{4}], \quad V[\frac{n}{2}+1 .. \frac{3n}{4}]$$

2. Compute new values for the elements of *RHS*

$$RHS(i) = LHS(2 * i) + V(i) \qquad 1 \leq i \leq \frac{n}{4}$$

3. Advance LHS, RHS, and VHS:

$$X[\frac{n}{2}+1 .. \frac{3n}{4}], \quad X[\frac{3n}{4}+1 .. \frac{7n}{8}], \quad V[\frac{3n}{4}+1 .. \frac{7n}{8}]$$

4. Go to step 2

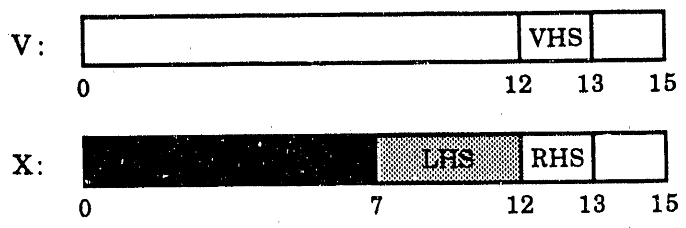


Figure 2 - Segmental Recomputation

Continue until *X* is exhausted. The algorithm comprises *Log n* iterative steps. Figure 2 depicts the algorithm's state after the first step. The computations at Step 2 are data independent and can execute in parallel. Moreover, since *LHS* and *RHS* are disjoint, *X* can be updated in place.

Since Sisal 1.2 lacks subarray operations, the *(Log n) RHS* vectors must be built independently and gathered into an array by the reduction operation

```
returns value of catenate rhs
```

The Sisal 1.2 code is

```
type OneDim = array[double_real]

function SegRec(X, V: OneDim returns OneDim)
  for initial
    n   := array_size(X) / 2;
    vhs := 0;
    rhs := array_adjust(X, 1, n)
  while n > 1 repeat
    n   := old n / 2;
    vhs := old vhs + old n;
    rhs := for i in 1, n
             returns array of
               old rhs[2 * i] + V[vhs + i]
           end for
  returns value of catenate rhs
  end for
end function % Segmental Recomputation
```

Two points about the code: 1) instead of forming the subvector *VHS* every iteration, we use a pointer, *vhs*, to point to the start of the subvector; and 2) observe that *LHS* is just ***old*** *RHS*.

The performance of this routine is dismal; in fact, regardless of the size of *X*, computing the new values for *X* one at a time is faster. The Sisal optimizers completely break down on this code. First, the runtime system allocates new memory for *rhs* every iteration, and deallocates ***old** rhs* every iteration. Second, the optimizers fail to recognize (and who can blame them) that *X* can be updated in place. Third, the build-in-place analysis fails to preallocate memory for the result. Thus, not only do we copy *rhs* every iteration as we move it into the result, we also copy the partial result every iteration as it grows and requires more space. The optimization discussed in the previous chapter (allocate two arrays outside the **for initial** expression and switch back and forth) alleviates the first problem, and an enhanced build-in-place analyzer solves the third problem. But unless *X* is updated in place, the cost of copying *rhs* every iteration will destroy performance. The computation per loop body (one addition) is too small to recuperate much, if any, of the cost.

The problem is not weak optimizers, but poor syntax. The lack of subarray operations destroys all hope of realizing that *X* can by updated-in-place. In Sisal 2.0, which includes subarray operations, the algorithm is clean, concise, and easily optimized for update-in-place. The Sisal 2.0 code is

```
type OneDim = array[double_real]

function SegRec(X, V: OneDim returns OneDim)
  let
    n := size(X) / 2;
    i := 1;
    j := n
  in
    while n > 1 do
      new n := n / 2;
      new i := j + 1;
      new j := j + n;
      new X := X[new i..new j:
                    X[i..j..2] + V[new i..new j]
    returns X
    end while
  end let
end function % Segmental Recomputation
```

Here we use *i* and *j* to point to the start and end of each subvector. The plus operation on Line 13 is an element-by-element vector addition. The update of *X* is now obvious. Moreover, it is easy for the compiler to realize that the subarray of *X* which is written, *X[**new** i.. **new** j]*, is disjoint from the subarray which is read, *X[i..j..2]*, since ***new** i := j + 1* (Line 10).

Conclusions

In this paper we discussed the definition of arrays in Sisal 1.2 and the changes to arrays in the new language definition, Sisal 2.0. The biggest changes are: true multidimensional arrays (flat arrays), array comprehensions, and subarray operations. We showed that these changes can yield more readable code and better performance. We are hopeful that the optimization techniques developed for Sisal 1.2 are extendable to the new language definition. The goal of any functional language must be to achieve equivalent, or better, performance than Fortran; otherwise, the language will not be widely accepted. Sisal 1.2 did achieve Fortran-like speed on shared-memory multiprocessors. We believe Sisal 2.0 can achieve even better performance.

Acknowledgements

I would like to thank David Cann for helping me attain and analyze the performance data presented in Section 3. We used the Alliant FX/80 at the Northeast Parallel Architectures Center. I would also like to thank David Cann and Tom DeBoni for proofreading this paper and making a number of useful suggestions. Finally, I would like to thank the organizing committee of the First International Workshop on Arrays, Functional Languages, and Parallel Systems for inviting me to speak.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

References

1. Cann, D. C. Compilation Techniques for High Performance Applicative Computation. Ph.D. thesis, Department of Computer Science, Colorado State University, 1989.

2. Culler, D. E. *Managing Parallelism and Resources in Scientific Dataflow Programs.* Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1989.

3. Feo, J. T., D. C. Cann and R. R. Oldehoeft. The Sisal Language Project. to appear *Journal of Parallel and Distributed Computing,* December 1990.

4. Gao G. R. and R. K. Yates. An Efficient Monolithic Array Constructor. in *Proceedings of the 3rd Workshop on Languages and Compilers for Parallel Computing,* Irvine, CA, August 1990.

5. Hudak, P. and A. Bloss. The aggregate update problem in functional programming systems. *Proc. Twelfth ACM Symposium on the Principles of Programming Languages.* ACM, New Orleans, LA, January 1985, pp. 300-313.

6. Hudak, P. and S. Anderson. *Haskell Solutions to the Language Session Problems at the 1988 Salishan High-Speed Computing Conference.* Yale University Technical Report YALEU/DCS/RR-627, Yale University, New Haven, CT, January 1988.

7. McGraw, J. R. et. al. *Sisal: Streams and iterations in a single-assignment language, Language Reference Manual, Version 1.1.* Lawrence Livermore National Laboratory Manual M-146, Lawrence Livermore National Laboratory, Livermore, CA, June 1983.

8. McGraw, J. R. et. al. *Sisal: Streams and iterations in a single-assignment language, Language Reference Manual, Version 1.2.* Lawrence Livermore National Laboratory Manual M-146 (Rev. 1), Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

9. McMahon, F. H. *Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range.* Lawrence Livermore National Laboratory Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.

10. Nikhil, R. S. and Arvind. *Id: a language with implicit parallelism.* Computations Structures Group Memo 305, Massachusetts Institute of Technology, Cambridge, MA, February 1990.

11. Oldehoeft, R. R. and D. C. Cann. Applicative parallelism on a shared-memory multiprocessor. *IEEE Software* **5,** 1 (January 1988), pp. 62-70.

12. Oldehoeft, R. R., D. C. Cann, et. al. *SISAL Language Manual, Version 2.0.* in preparation.

13. Ranelletti, J. E. Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages. Ph.D. thesis, Department of Computer Science, University of California at Davis/Livermore, 1987.

14. Skedzielewski, S. K. and R. J. Simpson. A simple method to remove reference counting in applicative programs. *Proc. ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,* Portland, OR, June 1989.

END

DATE FILMED

01 / 16 / 91