JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 10, 349–366 (1990)

A Report on the Sisal Language Project*

JOHN T. FEO AND DAVID C. CANN

Computer Research Group (L-306), Lawrence Livermore National Laboratory, P.O. Box 808, Livermore, California 94550

AND

RODNEY R. OLDEHOEFT

Department of Computer Science, Colorado State University, Fort Collins, Colorado 80523

Sisal (Streams and Iterations in Single Assignment Language) is a general-purpose applicative language intended for use on both conventional and novel multiprocessor systems. In this report we discuss the project's objectives, philosophy, and accomplishments and state our future plans. Four significant results of the Sisal project are compilation techniques for high-performance parallel applicative computation, a microtasking environment that supports dataflow on conventional shared-memory architectures, execution times comparable to those of Fortran, and cost-effective speedup on shared-memory multiprocessors.

1. INTRODUCTION

Sisal (Streams and Iterations in a Single Assignment Language) is a general-purpose applicative language intended for use on both conventional and novel multiprocessor systems. The project began as a collaborative effort by Lawrence Livermore National Laboratory, Colorado State University, University of Manchester, and Digital Equipment Corp.

Sisal, a derivative of Val [4], was defined in 1983 [28] and revised in 1985 [29]. Since then the language definition has not changed, providing a stable testbed for implementors, programming language researchers, and users. The project's objectives are:

1. define a general-purpose applicative language,
2. define a language-independent intermediate form for dataflow graphs,
3. develop optimization techniques for high-performance parallel applicative computing,
4. develop a microtasking environment that supports dataflow on conventional computer systems,
5. achieve execution performance comparable to that of Fortran, and
6. validate the applicative style of programming for large-scale scientific applications.

These goals address issues in programming languages, compilers, operating systems, performance, and software engineering. The first three are typical of dataflow projects. The last three set the Sisal effort apart—they reflect the computing environment at Lawrence Livermore National Laboratory and other government facilities. In particular, the focus on conventional systems is in sharp contrast to that of other dataflow language projects that assume hardware support. Our intention is to define a language and an intermediate form independent of architecture and then to develop code generators and run time systems for specific target machines. This approach has paid off handsomely, as Sisal is running today on uniprocessors, conventional shared-memory multiprocessors [26, 37], the Cray X/MP [27], the Warp [19], the Connection Machine [15], the Mac II [32], and a variety of dataflow machines [2, 20].

Why the interest in applicative languages at the national laboratories? Scientists there, and elsewhere, realize that the next generation of single-processor systems will not deliver the magnitude of increase in computing power that they require. They understand that they must parallelize their codes and move to multiprocessor systems. However, despite the availability of such systems, the number of scientific parallel applications in use today remains virtually zero. The reason: conventional parallel programming languages thwart programmer productivity and hinder analysis. They fail to separate problem specification and implementation, fail to emphasize modular design, and inherently hide data depen-

* This report was prepared as an account of work sponsored by the U.S. Government. Neither the United States nor the U.S. Department of Energy, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference to a company or product name does not imply approval or recommendation of the product by the University of California or the U.S. Department of Energy to the exclusion of others that may be suitable. The views, opinions, and/or findings contained in this report are those of the authors and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documents.

0743-7315/90 $3.00

dencies. Compilers that automatically parallelize programs written in sequential languages are a solution [38, 39], but we believe that these languages restrict the formulation of parallel algorithms and will always deter the automatic exploitation of parallel architectures [48].

Applicative languages provide an easy-to-use and clean parallel programming model that facilitates algorithm development and simplifies compilation. In an applicative program the value of any expression depends only on the values of its subexpressions and not on their order of evaluation. Applicative semantics only allow programmers to define data dependencies among operations. The scheduling of operations, the communication of data values, and the synchronization of concurrent operations are the responsibility of the run time system. The user does not (in fact cannot) manage these system operations. Relieved of the most onerous chores of parallel programming, the user is free to concentrate on algorithm design and application development.

Although ease of programming is important, at the national laboratories performance is the bottom line. If applicative languages are to gain adherents in the scientific community they must achieve execution speeds comparable to that of Fortran on state-of-the-art supercomputers. Since current dataflow machines are not capable of supercomputer performance, we have defined Sisal and IF1 [43] (our intermediate form) independent of architecture and have developed a run time system that supports dataflow on conventional shared-memory computer systems typical of today's supercomputers.

In Section 2, we discuss Sisal's characteristics and features. In Section 3, we describe the Sisal compiler and its optimization techniques for high-performance parallel applicative computing. In Section 4, we describe a microtasking environment that supports dataflow on conventional shared-memory computer systems. In Section 5, we show that Sisal programs can execute as fast as Fortran programs on conventional single-processor systems, plus automatically exploit conventional shared-memory multiprocessor systems. In Section 6, we discuss our future plans.

2. SISAL LANGUAGE SUMMARY

Sisal is a strongly typed, general-purpose applicative language that supports data types and operations for scientific computation and has a Pascal-like syntax for minimizing learning time and enhancing readability. Sisal has several important semantic properties. First, all functions are mathematically sound—there are no side effects. Second, Sisal programs are referentially transparent. Since names are bound to values and not memory locations, there is no aliasing. Third, the language is single assignment—a name is assigned a value only once. In general, a Sisal program defines a set of mathematical expressions where names stand for specific values, and computations progress without state. These properties make the transformation from source code to dataflow graph trivial.

TABLE I
Array Create Operations

Operations	Comments
`returns array of x`	Gather array
`array [1: 1, 2, 3]`	Create array
`array_fill(1, N, 0)`	Create array of 0's
`A[1: 0]`	Replace A[1] with 0
`A || B`	Catenate A and B

A Sisal program is a collection of separately compiled files called compilation units. Each compilation unit includes a list of declared function names visible outside the unit, a list of corresponding function definitions, and possibly the definitions of additional functions. A function can take zero or more arguments and must return one or more values. The type of each argument and result value is declared in the function's definition header. A function has access only to its arguments and there are no side effects. Each function invocation is independent; functions cannot retain state between invocations.

Sisal handles exceptions by producing special error values.[1] Although an efficient implementation may require special hardware, error values have three advantages:

(1) their presence alerts the user to an exception,
(2) they permit the computation to continue, and
(3) correct data are preserved to the extent possible.

The value **error** is a proper element of every Sisal type. Conditions that produce errors include underflow, overflow, divide by zero, subscript out of bounds, and conditional expressions whose test clauses are in error. The semantics of the language defines rules for propagating error values. Every effort is made to preserve data not in error. For example, an array is **error** if it includes error values, but the elements of the array are still accessible.

Sisal includes the standard scalar data types: boolean, character, integer, real, and double precision. It also includes the aggregate types: array, record, union, and stream. All arrays are one-dimensional. Multidimensional arrays are defined as arrays of arrays. The type, size, and lower bound of an array are a function of execution. The components of a multidimensional array may have different lengths and lower bounds, resulting in jagged arrays. Arrays are created by gathering component elements, "modifying" existing arrays, or catenation. Table I gives examples of each. The ex-

[1] Errors are supported in the interpreter and the native code compiler being developed at Lawrence Livermore National Laboratory; they are not supported in Osc, the native code compiler developed at Colorado State University and discussed in this paper.

TABLE II

Stream Operations

Operations	Comments
`stream_append(A, B)`	Append B to A
`stream_first (A)`	First element of A
`stream_rest (A)`	Tail of A
`stream_empty (A)`	Empty stream?
`A || B`	Catenate A and B

plicit, concise form of these operations greatly simplifies memory preallocation and update-in-place analysis.

A stream is a sequence of values of uniform type. In Sisal, stream elements are accessible in order only; there is no random access to elements. Table II lists the stream operations in Sisal. A stream can have only one producer, but any number of consumers. By definition, Sisal streams are nonstrict—each element is available as soon as it is produced. Run time systems must support the concurrent execution of a stream's producer and consumers. As such, streams can express pipelined parallelism and are a natural medium for program input and output.

Sisal supports both sequential and parallel loops. The **for initial** expression, illustrated in Fig. 1, resembles sequential iteration in conventional languages, but retains single-assignment semantics. It comprises four segments: initialization, loop body, termination test, and result 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, possibly on the basis of their previous values. The rebinding of loop names to values is implicit and occurs between iterations. Loop names prefixed with **old** refer to previous values. The termination test may appear either before or after the body. If it appears before, the body might not execute; if it appears after, the body will execute at least once. The returns clause defines the results and arity of the expression. Each result is either the final value of some loop name or a reduction of the values assigned to a loop name during loop execution. Sisal supports seven intrinsic reductions: **array of, stream of, catenate, sum, product, least**, and **greatest.** The order of reduction is determinate.

The **for** expression, illustrated in Fig. 2, provides a means for specifying independent iterations. The semantics of the expression does not allow references to values defined in other iterations. The **for** expression comprises three parts: a range generator, a loop body, and a returns clause. The range generator is a dot or cross product of a set of sequences or scatters (see Table III). An instance of the loop body is executed for each index, value, or n-tuple. The returns clause defines the results and arity of the expression. Each result is a reduction of values defined in the loop body (**array of, stream of, catenate, sum, product, least**, or **greatest**). The range generator specifies the order of reduction and defines the size and structure of aggregate objects. For example, the expression

```
for i in 1,n cross j in 1,m
returns array of (i + j)
end for
```

returns a two-dimensional array of n rows and m columns. At first, many Sisal programmers fail to understand the subtleties of this syntax. A common mistake is to write the transpose of an $(n \times m)$ matrix as

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

The correct expression is

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

To allow for mutually recursive functions and to encourage modular design Sisal includes **global** and **forward** function definitions. Although Sisal permits recursive function definitions, it does not permit recursive definitions of value names. The compiler enforces a strict *definition before use* policy on all value names. One consequence of this policy is that users must specify the order in which elements of recursive aggregates are computed. Consider the array definition

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

```
for  initial
     i := 1;
     x := Y[1]
while i < n repeat
     i := old i + 1;
     x := old x + Y[i]
returns  array  of  x
end  for
```

FIG. 1. The **for initial** expression.

```
for i in 1, N
    x := A[i] * B[i]
returns value of sum x
end for
```

FIG. 2. The **for** expression.

The Sisal expression

```
X := for i in 1, n cross j in 1, n
  returns array of
     if (i = 1) | (j = 1) then
       1
     else
       X[i, j - 1] + X[i - 1, j]
     end if
   end for
```

is illegal. A legal Sisal expression for Eq. (1) 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
      ends for
  returns array of row
  end for
```

Note that the order of computation is explicit. In a functional language that permits recursive definitions, such as Id [31], the equivalent expression is

```
X = makearray (1, n)
  { if (i == 1) or (j == 1) then
      1
    else
      X[i, j - 1] + X[i - 1, j]
  }
```

Here the order of computation is implicit, resolved at run time by the availability of results.

Although it obscures parallelism at the source level, excluding recursive definitions simplifies language implementation. First, we do not need special hardware such as full-and-empty bits to delay reads at run time. Second, we do not need sophisticated analysis routines or special hardware to detect recursive definitions that deadlock. Third, the system can bound parallelism without restriction because deadlock is impossible. On the matter of clarity, consider Eq. (1). If we maximize parallelism, the computation sweeps across the matrix from the top left-hand corner to the bottom right-hand corner in a diagonal wave—the computations along a diagonal are data independent and can execute in parallel. But the parallel nature of the expression is not apparent from the Sisal code. However, if the Sisal run time system schedules all the loop bodies simultaneously and has each wait for its inputs, we will realize the parallelism hidden in the source code.

Strict adherence to applicative semantics may introduce substantial execution costs, possibly negating the effects of parallelism. Consider the Sisal expression

```
X := Y[1: 3]
```

Strict adherence to single-assignment semantics requires construction of a new array identical to Y except at index location 1. If the expression is the last consumer of Y, the copy is unnecessary. Worse yet consider the Sisal expression

```
for initial
  i := 1;
  n := 50000;
  A := array[1: 1]
while i < n repeat
  i := old i + 1;
  A := array_addh(old A, i + old i)
returns value of A
end for
```

A strict applicative implementation of the *array_addh* operation requires construction of a new array each iteration, resulting in the copy of $O(n^2)$ values. On the Sequent Balance 21000, for $n = 50{,}000$, the Sisal expression executes in about 1 h, while the equivalent Fortran code

```
  integer A(50000)
  do 5 i = 1, 50000
5     A(i) = i
```

executes in less than half a second. Since there is only one

TABLE III

Range Generators

Operations	Comments
`for x in A`	A scatter
`for i in 1, n`	A sequence
`for x in A dot y in B`	A dot product of two scatters
`for i in 1, n cross j in 1, m`	The cross product of two sequences

consumer of A, the *array_addh* operation on the next iteration, the copying is superfluous. In the next section we present analysis procedures that identify these and other instances where copy operations are not needed to maintain referential transparency.

In the last example, even if we build the array in place, the run time system may still have to copy the array every iteration to find room for the new elements. Again, in the worst case, we will copy $O(n^2)$ values. We can eliminate the copying by calculating the size of the final array and preallocating memory. **For** expressions can also generate extraneous copying when constructing aggregates. Since the loop bodies are data independent, they may be executed by independent processors. On completion of the expression, the run time system will have to gather (copy) the partial results. In the next section we describe compile time techniques that insert operations into the code to compute the size of most aggregates and preallocate memory.

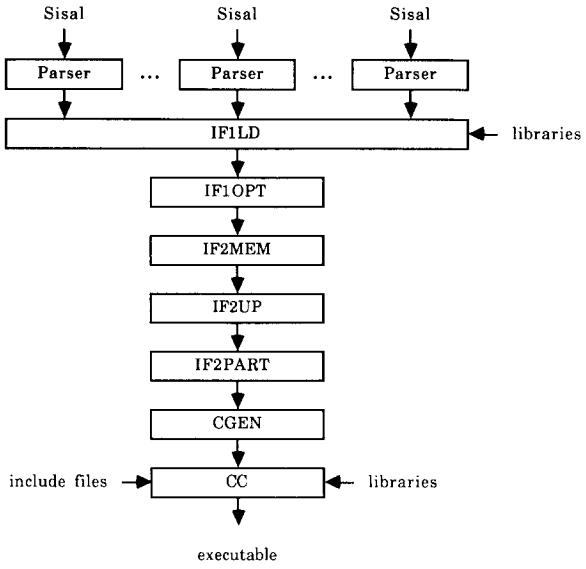


FIG. 3. Sisal language processing.

3. THE COMPILER

In this section we present an overview of our current Sisal compiler (called Osc) and provide a brief enumeration and illustration of its optimization subphases. Osc successfully eliminates unnecessary copying and greatly reduces storage management overhead for most Sisal programs. In this discussion, we assume that the reader has a working knowledge of reference counting and its use in storage management [12] and dataflow graphs and their use in compilation [18, 25]. We also assume that the reader has an understanding of conventional optimization techniques [5], including interprocedural analysis [6].

Osc, an extensive rework of a prototype developed for the Hep multiprocessor [35], is a state-of-the-art optimizing compiler. Figure 3 diagrams its phases and subphases of operation. The front end compiles Sisal source into IF1 [43], an intermediate form defining dataflow graphs adhering to applicative semantics. An IF1 program consists of one or more acyclic graphs made up of simple nodes, compound nodes, graph nodes, edges, and types. Nodes denote operations, edges transmit data between nodes, and types describe the transmitted data. Simple nodes represent operations such as addition, division, and array and stream manipulation. Compound nodes encapsulate one or more subgraphs to define structured expressions such as conditionals and **for initial** and **for** expressions.

Because the production of quality code requires complete information, the second phase of compilation (IF1LD) forms a monolithic IF1 program. The monolith is then read by a machine-independent optimizer (IF1OPT) that applies conventional optimizations such as function expansion, invariant code removal, common subexpression elimination, constant folding, loop fusion, and dead code removal. Except when presented with recursive calls or user directives, all functions, by default, are in-lined to form a single dataflow graph. The common subexpression eliminator looks outside branches of conditional statements in an attempt to find more common subexpressions. See [45] and [10] for further discussion of these optimizations.

After the machine-independent optimizations, a *build-in-place* analyzer (IF2MEM) preallocates array storage where compile time analysis or run time expressions can calculate their sizes [41]. The result of this analysis is a semantically equivalent program graph in IF2 [47]. IF2 is a superset of IF1, but is not applicative because it supports operations that directly reference and manipulate "abstract" memory (called **AT-nodes**). The next phase of compilation (IF2UP) does *update-in-place* analysis [10]. The analyzer restructures some graphs, while preserving program correctness, to help identify (at compile time) operations that can execute in place and to improve chances for in-place operation at run time when analysis fails. The analysis routines are based on work done at Yale University [21]. We discuss these two phases in detail in the next two subsections.

After update-in-place analysis, we invoke a parallellizer called IF2PART to define the desired granularity of parallelism. The analysis is based on estimates of execution time and other parameters. The user can change these parameters at compile time to have some control over the parallelization. Currently, IF2PART selects only **for** expressions and stream producers and consumers for parallel execution. Only **for** expressions with estimated costs greater than a threshold and nested no deeper than a defined parallel nesting level (the default being all levels) are selected. We do not exploit function-level parallelism because our experience has shown that most nonparallel functions are too small to justify the overhead of task creation and synchronization. Note that a func-

tion's parallel subtasks (namely, the included **for** expressions and stream producers and consumers) will be selected for parallel execution. IF2PART is a simplification of work by Sarkar [42]. Many experiments and possible elaborations await future work, especially as we target more unusual architectures.

In the last phase, CGEN translates the optimized IF2 graphs into C code, which is then compiled using the local C compiler to produce an executable program. Preprocessor directives provide the definition of target-dependent operations and values. Library software, linked during this phase of compilation, provides support for parallel execution, storage management, and interaction with the user. We chose C as an intermediate form to shorten development time, increase system portability, and allow experimentation with future optimizations by manual editing. However, the quality of the available C compiler can limit overall performance; hence, for the Sequent Balance we wrote a simple machine-dependent optimizer, which works at the assembly language level, to improve register utilization and reduce code size.

3.1. Build-in-Place Analysis

This optimization, IF2MEM, attacks the incremental construction problem introduced in the previous section. The algorithm is two-pass in nature. Pass 1 visits nodes in dataflow order and builds, where possible, expressions to calculate array sizes at run time. These expressions are IF1 code fragments; they are inserted in the graphs before the nodes producing the arrays whose sizes they define. We call these nodes **potential AT-nodes**. Their definition is a function of the semantics of the array constructor and the size expressions of its inputs. Determining the size of an array built during loop execution requires an expression to calculate the number of loop iterations before the loop executes. Deriving this expression is not possible for all loops. The final function of pass 1 is to push, in the order of encounter, the **potential AT-nodes** onto an **AT-node** conversion stack. This stack drives the second pass of the algorithm.

Pass 2, considering only **potential AT-nodes**, inserts nodes for memory preallocation and manipulation, converts the **potential AT-nodes** to **AT-nodes**, and appropriately wires memory references among them (inserting edges transmitting pointers to memory). If the node under consideration is the parent of an already converted node, it can build its result directly into the memory allocated to its child, thus eliminating the intermediate array. If the node is not the parent of an already converted node, it must build its result in a "new" memory location. Note that the ordering of nodes in the **AT-node** stack guarantees processing of children before parents. As a final responsibility, pass 2 must add **P** mark pragmas (edge annotations) to those edges carrying arrays built in place. These annotations specify that arguments to array constructors were built in place.

Consider the Sisal expression

```
A || array_fill (1,N,0)
```

and its unoptimized IF1 graph

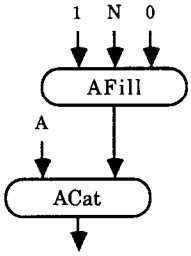


Figure 4 is the result of applying IF2MEM to the expression (assuming $N \geq 1$). In pass 1, the **AFill** node and then the **ACat** node are assigned size expressions and pushed onto the **AT-node** conversion stack. The two size expressions are N and $array\_size(A) + N$, respectively. In pass 2, the nodes are considered in reverse order. Because it is not the parent of an **AT-node**, the **ACat** node requires new memory. IF2MEM builds a graph fragment to allocate this memory (using **ACat**'s size expression), changes the **ACat** node into an **ACatAT** node, and builds an edge from the graph fragment to the node. Next, the **AFill** is popped from the **AT-node** conversion stack. Since it is now the parent of an **AT-node** child (the **ACatAT** node), IF2MEM does not build a fragment to allocate storage for its result, but instead builds a code fragment to derive the location of its result in the storage already allocated to its child. The fragment shifts the base address of the storage by the size expression associated with the first input to the **ACatAT** node: $array\_size(A)$.

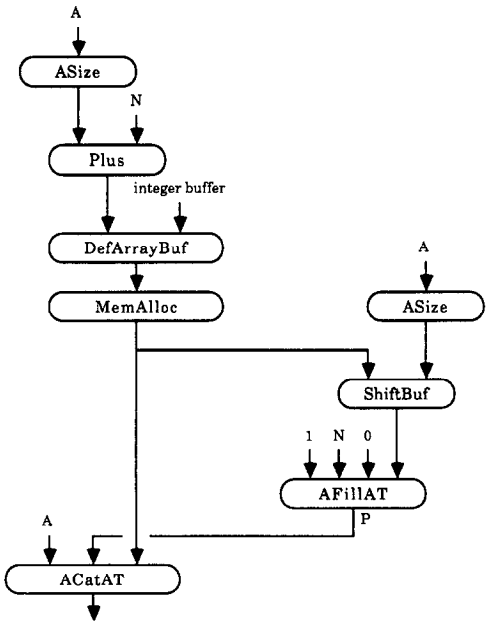


FIG. 4. Optimized IF2 graph for memory preallocation.

Pass 2 completes by converting **AFill** to **AFillAT**, wiring a reference from the shifted address to the node, and placing a **P** mark on the edge linking the converted nodes (communicating that the **AFillAT** node built its result in-place).

3.2. Update-in-Place Analysis

This optimization, IF2UP, attacks the aggregate update problem introduced in the previous section. Also, it serves to enhance the build-in-place analysis described above. That is, IF2UP treats an array as two separately reference counted objects: a dope vector defining the array's logical extent and the physical space containing its constituents. IF2MEM only produces IF2 computations to preallocate physical space and direct the placement of constituents; it does not preallocate dope vectors or optimize their access. As subcomputations place constituents in the preallocated memory, dope vectors cycle between dependent nodes to communicate the current status of the regions under construction. As a result, multiple dope vectors may reference the same or different regions of the physical space and the individual participants in the construction will produce new dope vectors to communicate the current status of the regions to their predecessors. Without update-in-place optimization, each stage in construction will copy a dope vector.

As an additional benefit, IF2UP eliminates most reference count operations. Reference counting can be a source of wasted computer time and parallel bottlenecks [37]. A prototype reference count eliminator developed at LLNL on the average elminated 67% of reference count operations in Sisal programs while opting, as a priority, to preserve parallelism among operations [46]. In contrast, IF2UP eliminates up to 98% of explicit reference counting in larger programs, ordering nodes using artificial dependence edges without regard for lost parallelism [8]. In practice this loss in parallelism potential has been small and of no effect in current implementations [10]. As a result, the inefficiencies of reference counting largely disappear, but the mechanism is preserved. Moreover, the few remaining occurrences do not merit special hardware support.

IF2UP also recognizes single-consumer streams to allow generation of support code with fewer critical sections. This is important because, in the most general instance, a stream may be the result of catenating many streams, each with its own producer, and there may be many consumers of the entire stream or substreams. The run time support for such an object is extensive. Elements are in linked lists attached to stream control blocks that are themselves linked because of catenation operations, and each stream element needs a reference count since there are multiple consumers. However, programmers usually do not use concatenation to form streams, and most streams have only one consumer. Here a simple circular buffer with minimal synchronization between the producer and the consumer suffices.

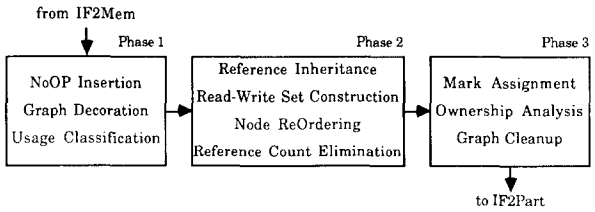


FIG. 5. The internal operation of IF2Up.

Figure 5 diagrams the internal operation of IF2UP. It takes as input an IF2 program and produces a semantically equivalent IF2 program. Phase 1 (subphases 1, 2, and 3) prepares each graph in the program for analysis. Phase 2 (subphases 4, 5, 6, and 7) eliminates unnecessary reference count operations. Phase 3 (subphases 8, 9, and 10) eliminates unnecessary copy operations and identifies single-consumer streams. The subphases operate as follows:

1. **NoOp** *insertion.* Here we simply insert **NoOp** nodes (data duplicators) to decouple copy logic from all nodes modifying aggregates. This isolates copying to a single node type to simplify later analysis. After this subphase, all modifiers work in place and the inserted **NoOps** unconditionally copy the consumed aggregates. The goal of the remaining subphases is to identify the unnecessary **NoOp** nodes.

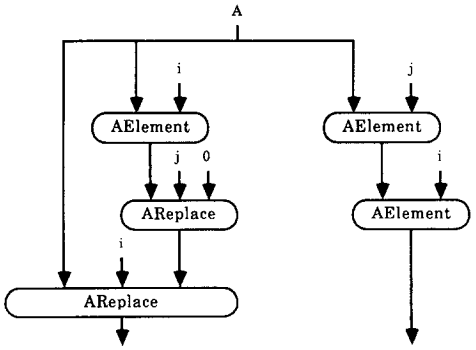
2. *Graph decoration.* This subphase annotates edges transmitting aggregates with pragmas that explicitly express a program's worst-case reference count behavior. The assignments naively assume that all nodes will execute in parallel, which requires that all consumers reference count their inputs. The three reference count pragmas of interest are **sr** for setting counts, **pm** for incrementing counts, and **cm** for decrementing counts.

3. *Usage classification.* Here we classify each edge transmitting aggregates as either write or read depending on destination node semantics and port of entry. A write-classified edge, in contrast to a read-classified edge, transmits aggregates destined for in-place modification (provided conditions guarantee correctness) or placement within other aggregates. This subphase assigns **W** marks to write-classified edges and leaves read-classified edges unmarked. To classify usage across graph boundaries, we examine functions in topological order and traverse graphs bottom up. This allows classification of actual arguments based on formal argument classifications and classification of compound node inputs based on subgraph usages.

Consider the Sisal function

```
type TwoDim = array[array[integer]];
function Main (i,j: integer; A: TwoDim
                returns TwoDim, integer)
  A[i,j: 0.0], A[j,i]
end function
```

and its unoptimized IF1 graph



Without optimization, both replace operations might require copying. Figure 6 shows the program graph after the first three subphases. Subphase 1 inserted two **NoOp** nodes and marked the replace operations to execute in place, as indicated by the **RO** marks. Subphase 2 annotated the graph with the **pm, cm,** and **sr** pragmas. Subphase 3 marked the edges carrying aggregates to write operations with the **W** marks.

4. *Reference inheritance.* This subphase eliminates implicit reference counting[2] where safe, to improve run time opportunities for copy avoidance when modifying inner dimensions of nested arrays. In unoptimized form, a replacement operation implicitly decrements the reference count of the constituent it replaces (assuming the constituent is an aggregate). However, preserving this constituent until the

[2] Implicit reference count operations are implicit in the management of nested aggregates and are not explicitly represented in the program graph.

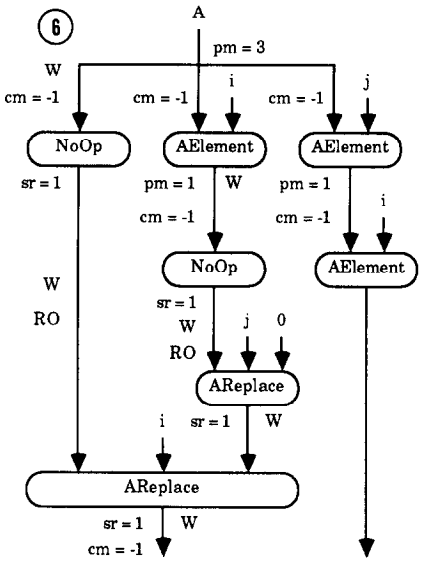


FIG. 6. Example program after subphase 3.

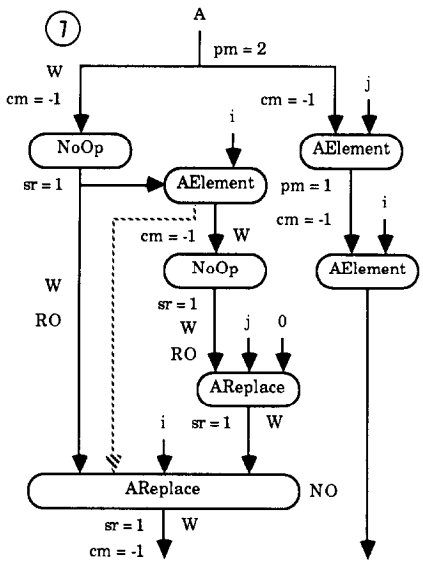


FIG. 7. Example program after subphase 4.

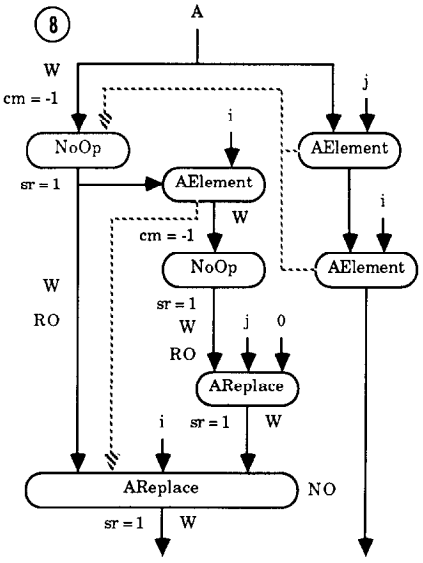


FIG. 8. Example program after subphase 7.

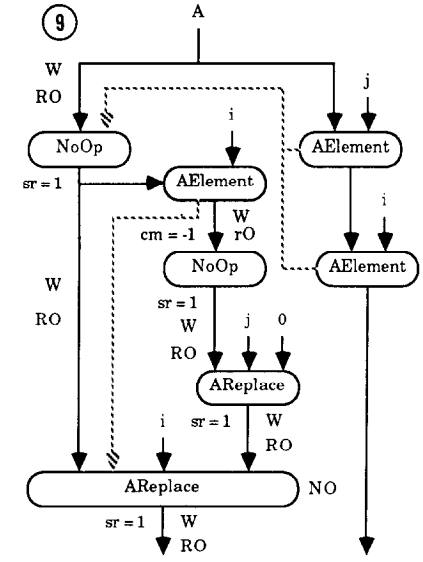


FIG. 9. Example program after subphase 10.

time of replacement can sometimes cause copying. During reference inheritance we restructure some replacement operations to give the implicit references to operations working on the replaced constituents.

5. *Read–write set construction.* Here to simplify the remaining subphases in phase 2, we build sets summarizing the usage of aggregates and their constituents. For each aggregate output of a node we build a local read and write set pair. The former identifies the read-classified edges attached to the output and the latter identifies the write-classified edges attached to the output. Then for each aggregate output that is either not the result of a dereference node or part of a reference inheritance transformation, we build a global read and write set pair. A global read set identifies both the output's immediate read-classified references and those of its constituents. Similarly, a global write set identifies both the output's write-classified references and those of its constituents.

6. *Node reordering.* This subphase defines a new partial ordering that maximizes opportunities for reference count and copy elimination without violating the original data dependencies. Where possible, we insert artificial dependence edges, without concern for lost parallelism, to force readers of an aggregate and its constituents to execute before modifiers of the aggregate and its constituents. The previously allocated global read and write sets drive this subphase of analysis.

7. *Reference count elimination.* This subphase proceeds in two independent steps. The first, called *phantom elimination,* eliminates reference counting that unnecessarily preserves aggregates across graph boundaries—we need only preserve aggregates imported to two or more graphs. The second, called *edge neutralization,* eliminates reference count pragmas on inputs to some read operations on the basis of the execution order derived during node reordering. As in the previous subphase, this step is a function of the usage sets. Consider node **R** that reads aggregate **A** transmitted by edge **E**. The analyzer removes the **cm** pragma annotating **E** if an artificial dependence edge occurs between **R** and a member of **A**'s local write set. It also erases the pragma if an artificial dependence edge occurs between **R** and all members of **A**'s global write set. If **A**'s global write set is empty, then **A** is, or is part of, a read only aggregate and does not require reference counting. In removing **E**'s **cm** pragma, the analyzer decrements the value associated with its antagonistic **pm** or **sr** pragma.

To continue the example begun above, Figs. 7 and 8 show the program graph after reference inheritance and the remaining subphases of phase 2, respectively. The dashed arrows represent artificial dependence edges. The **NO** pragma on the outermost **AReplace** node instructs it not to decrement the reference count of the replaced row. Analysis gave this reference to the **AElement** node dereferencing the *i*th row to improve the chances for in-place modification of the row at run time.

8. *Mark assignment.* Here we assign mark pragmas defining data access rights and drive them across graph boundaries and through function graphs and subgraphs of compound nodes. Intuitively, this subphase partially interprets a reference count optimized program, not to realize execution, but to derive information about aggregate mutability and appropriately record it in the graphs. We use **R** marks to annotate edges known to transmit *mutable* aggregates and **r** marks to annotate edges known to transmit *potentially mutable* aggregates. In the case of arrays and streams, the **R** and **r** marks apply only to dope vectors. We use **O** marks to annotate edges known to transmit arrays with mutable physical space and streams with a single consumer. We use **unknown** marks to annotate edges known to transmit *immutable* data; this is equivalent to unmarked edges.

This phase of the optimization begins at the program entry point and visits all nodes in dataflow order. All call paths are followed except those that form a cycle. The assigned marks are a function of an edge's reference count pragmas, source node semantics, and source node input marks. With regard to functions, if two or more call sites propagate different marks to the same formal argument, we label the callee *unstable.* After examining all call paths to an unstable function entry point, we remark the arguments causing the instability with **r** marks. The analyzer handles **for initial** expressions similarly; it remarks with an **r** any loop-carried value whose initial mark is different from the mark assigned by the redefinition of the value in the body. We assume that arguments to the main function from the outside world are mutable. Note that for simplicity, this subphase of analysis assumes that physical space of an array or stream dereferenced from another aggregate is immutable.

9. *Ownership analysis.* This subphase attempts to compensate for the previous phases' inability to assign **O** marks to edges transmitting extracted arrays or streams. Further, this subphase identifies streams with single consumers. In the current implementation, however, the analysis is conservative. This subphase only assigns an **O** mark to an array if it can assign it to all arrays in the program. Similarly, it only identifies a stream as having a single consumer if it can identify that all streams in the program have single consumers. To assign **O** marks to all arrays in a program, the analyzer attempts to verify that during execution the reference count of each array's physical space will remain at one. Because this is a function of dope vector copying, which is the only means of incrementing physical space reference counts, the analyzer need only analyze the marks assigned to **NoOp** nodes. If those that copy only dope vectors have input edges with **R** marks, then all arrays in the program will have mutable physical space at run time. Similar analysis identifies single-consumer streams.

10. *Graph cleanup.* This subphase simply removes **cm** pragmas annotating the inputs of unnecessary **NoOp** nodes.

Completing our example, Fig. 9 shows the final program graph. Because the outermost **NoOp** is now the final consumer of **A**, it is marked to execute in place (the **RO** pragma). On the other hand, because of the possibility of row sharing, subphases 8 and 9 marked the innermost **NoOp** for run time copy avoidance (the **rO** mark).

3.3. Code Generation

CGEN translates optimized IF2 graphs into C code. With two exceptions, each IF2 function graph is directly mapped into an equivalent C function. The first exception concerns the **for** expressions selected for parallel execution by IF2PART. Here CGEN removes each selected expression from its enclosing computation, leaving in its place a run time system call to instantiate its parallel execution. The body of each expression is then compiled into a generic function, which takes iteration bounds as arguments. The second exception concerns expressions that produce and consume streams. CGEN also removes these expression from their enclosing computations, leaving run time system calls to instantiate their parallel execution. The compiler then packages them as separate functions. In both of the above cases, synchronization primitives are added to coordinate parallel execution.

It is the responsibility of CGEN to recognize the pragmas inserted during IF2 optimization and generate the appropriate code. Reference count operations, where required, are compiled directly into the resulting C code. Memory preallocation operations are compiled into dynamic storage allocation requests and pointer manipulations.

4. THE RUN TIME SYSTEM

The run time software supports the parallel execution of Sisal programs, provides general-purpose dynamic storage allocation, implements operations on major data structures, and interfaces with the operating system for input/output and command line processing. The system was first described in [33] and considerable evolution has since occurred.

Sisal run time support makes modest demands of the host operating system. Support for parallel execution is in the form of threads or lightweight processes (similar to those provided by the Mach operating system [3]). The run time system maintains two queues of executable tasks: the *Ready List* and the *For Pool.* Execution begins at the function *Main.* At program initiation, run time options and the formal parameters of *Main* are read from the command line. For non-stream values, inputs are read at initiation and results are written at termination in Fibre format [44], a text form that describes scalar and structured Sisal values. For stream parameters and results, associations are made with files, and special stream producing (input) and consuming (output) threads are added to the *Ready List* for processing during execution. The run time system allocates stacks for threads on demand, but every effort is made to reuse previously allocated stacks and thus reduce allocation and deallocation overhead. Stack overflow is not monitored, however, and can result in program termination. The programmer can use a run time option to adjust stack size when anticipating overflow. The weakness of Sisal input/output is in the construction of simple interactive Sisal programs; solutions are under investigation.

4.1. Threads

At program initiation a command line option specifies the number of operating system processes to be instantiated for the duration of the program. These processes, called workers, are constant in number and look for work to do in the form of threads, whose number varies over the program's execution. While general-purpose thread support is available from other sources (see [7], for example), our thread management subsystem is optimized for Sisal and does not rely on vendor-supplied software.

Figures 10, 11, and 12 show the various activities a worker may engage in as it seeks work provided by Sisal execution and identify the major data structures needed to support parallel execution. A worker persists in one of three modes of operation, depending on whether it has recently handled *For* work, *Ready List* work, or is idle because neither kind of work is available.

4.1.1. *For Pool Threads*

A worker examines the *For Pool* for a piece of a **for** expression to execute. A "**for** slice" (one or more consecutive **for** bodies) is obtained and the thread stack already held by the worker is used to execute the code for the slice. Now one of three events can occur:

1. The slice may terminate normally. If the slices of this **for** expression are depleted, the worker places the thread in which the **for** expression occurs on the *Ready List* and returns to the *For Pool*; otherwise, the worker returns to the *For Pool* for another slice of the expression. By persisting in executing **for** slices, a worker avoids deallocating and allocating stacks in which to execute.

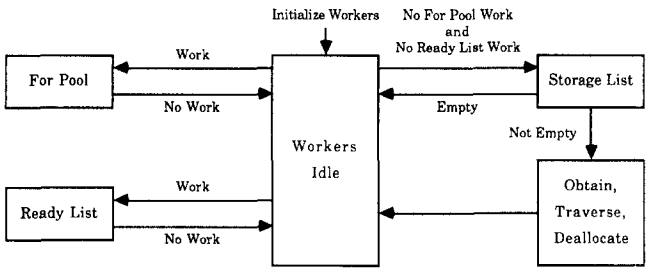


FIG. 10. Worker's state diagram.

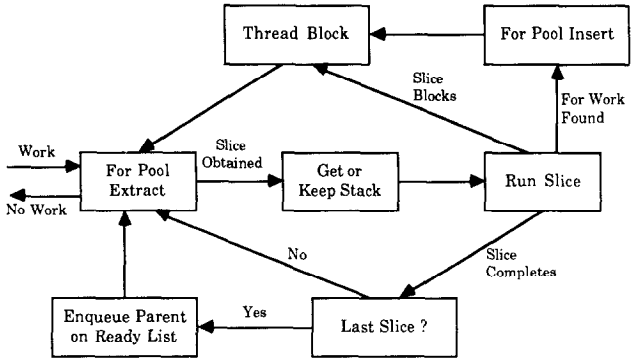


FIG. 11. For pool state diagram.

2. More work (a nested **for** expression) may be encountered. A new entry is made in the *For Pool*, containing these values:

a. the address of the loop,
b. the initial lower bound,
c. the upper bound,
d. the size of the **for** slice, and
e. the address of the descriptor for the thread in which the **for** expression occurs.

Depending on a compile time option, the size of the **for** slice is either constant or specified by "guided self-scheduling" [40]. The original **for** expression becomes a blocked thread to be reactivated when the new **for** expression completes.

3. The slice may block because dynamic storage is unavailable, in which case the slice becomes a blocked thread and the worker returns to the *For Pool* for more work.

4.1.2. *Ready List Threads*

Threads appear on the *Ready List* as a result of being made executable by other events. A **for** expression may complete, enabling the thread in which it appears to become runnable, or storage may become available, enabling a thread that blocked due to lack of storage to run. At program initiation, a single thread for the main function is placed on the *Ready List*. All *Ready List* entries have allocated thread stacks attached, so a worker in this mode of processing does not need its own stack. Upon obtaining a thread, execution begins. Once again, three events are possible:

1. If the thread terminates normally, the worker checks the parent context thread to see if it is waiting only for this thread to complete. If it is, the worker places the parent thread on the *Ready List*, deallocates the obsolete stack, and returns to the *Ready List* for more work; otherwise, it deallocates the obsolete stack and returns to the *Ready List*.

2. A **for** expression may be encountered. The worker handles it in a fashion similar to that of *For Pool* processing. Note that the worker must return to the *Ready List* for more work since it has no stack of its own.

3. The thread may block, in which case the worker returns to the *Ready List* for more work.

4.1.3. *Overlapped Storage Deallocation*

If a worker in either of the preceding modes finds no work to do, it returns to the central state shown in Fig. 10. If neither *For Pool* nor *Ready List* threads are available, the worker examines the *Storage Deallocation List* for deferred storage deallocation work. This list has entries for hierarchically structured aggregates, such as multidimensional arrays, which are no longer in use. If these structures were deallocated as soon as they were not needed, we could potentially introduce sequential code sections in parallel constructs, degrading parallelism. See [9] for an example of this phenomenon. Moreover, by deferring the traversal and deallocation of hierarchically structured aggregates, a program may complete without doing it at all. If unavailable storage idles enough threads, at least one worker will eventually idle and deallocate the needed space. Currently, overlapped storage deallocation is not implemented.

4.2. Storage Management

Sisal relies on dynamic storage allocation for many data values, such as arrays and streams, and for internal objects, such as thread descriptors and execution stacks. To support this, we needed a mechanism that was fast, efficient, and parallelizable. A two-level method that has proven satisfactory evolved.

4.2.1. *Parallel Boundary Tag Method*

The standard boundary tag scheme [22] was augmented with multiple entry points to a circular list of free blocks. We have an array of pointers, each addressing a zero-size free block on the list. A worker selects a pointer on the basis of its own integer identifier, thereby spreading list entry contention across the array. The free list search was parallelized, even though blocks are sometimes completely removed from

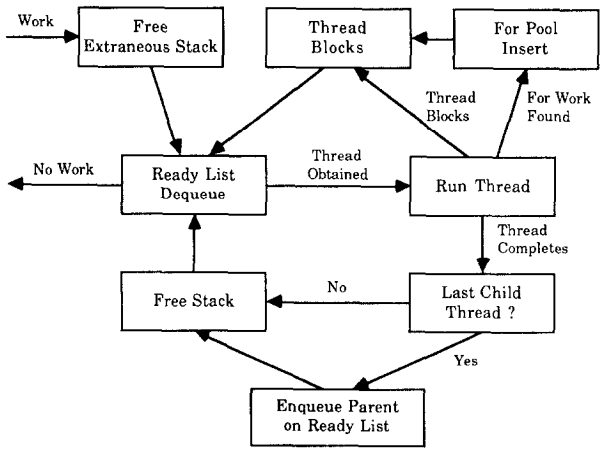


FIG. 12. Ready list state diagram.

the list. Likewise, deallocation involves coalescing physically adjacent but logically distant blocks in parallel.

4.2.2. *Exact Fit Caching Using Working Sets*

To increase the speed of allocation and deallocation, an exact fit caching mechanism was interposed before the boundary tag scheme for most storage operations [34]. It uses a working set of different sizes of recently freed blocks (blocks of the same size are chained in a sublist). As in the working set method for virtual memory management [14], a "clock" which ticks at each allocation is compared with a time stamp stored when each size is allocated or deallocated. This occurs as an allocation request looks for an exact fit and finds a mismatch. If a size is too "old" (the definition of "old" is a run time option), all blocks in the sublist are recycled to the boundary tag pool. If an exact fit is found, that block is unlinked, and the time stamp is updated in the new first sublist block. If this was the only block of this size, the list of differently sized blocks reduces in size by one. If no exact fit is found, an allocation occurs from the boundary tag pool. Our first implementation used a single, shared exact-fit cache for all workers. To eliminate synchronization overhead, each worker now has its own cache. This results in faster execution, but can cause exact-fit misses if the size desired is in another worker's cache.

4.2.3. Storage Deadlock

Contention among threads for dynamically allocated storage is the only source of possible deadlock in Sisal. A thread that cannot obtain needed storage becomes blocked and is reactivated when later deallocations occur. The last worker to become idle because there is no work to do checks for the presence of threads blocked for storage. It empties all the exact-fit caches to maximize boundary tag coalescing and seeks one or more threads to place on the *Ready List*. If this fails, the program stops with a deadlock message. A run time option specifies the total amount of storage to manage, and a larger value usually solves the problem.

4.3. Run Time Support for Arrays

Arrays in Sisal are one-dimensional data values of elements of the same type. Our implementation places elements in contiguous, dynamically allocated storage, sometimes followed by space for dynamic growth. Arrays are strict in this implementation; a thread containing a **for** expression that produces an array blocks until that expression has been processed. Since arrays are dynamic in size, bookkeeping information is kept in a *dope vector:*

# of elements	Pointer to physical space	Pointer to first element	Reference Count

Multidimensional arrays are arrays of dope vectors pointing to other such arrays; the last dimension is a one-dimensional array of base type. The layout of the physical space is

# of bytes	Reference Count	Expansion Count	Size of free space	Array elements	Free space

Both the dope vector and the physical space require reference counts because more than one thread may have access to the dope vector and more than one dope vector may point to the physical space (subarrays of the original). In Section 2 we explained that arrays can grow during execution. Although build-in-place analysis can compute the final size of most arrays and preallocate storage, in those cases where it cannot, we reduce copying by allocating *free space*. The size of the extra storage is the product of a run time option value and the number of times the array has been copied owing to expansion.

4.4. Run Time Support for Streams

In Sisal streams must be nonstrict. Typically an iteration produces values, and another iteration consumes them. As already mentioned, to implement parallelism, stream producing and consuming iterations are packaged by the Sisal compiler as sequential loops in separate threads. They are instantiated with stacks and placed on the *Ready List* for processing by workers. If a consumer thread attempts to use a value not yet produced, it blocks. Likewise, to prevent excessive consumption of memory, a producer thread blocks if it gets some number of elements ahead of the slowest consumer. To prevent continuous blocking and unblocking, producers and consumers awake only after some number of values have been consumed or produced, respectively; these numbers are run time options with defaults.

In the most general case, a stream value may be a concatenation of many constituent streams, each with its own producer thread and one or more consumer threads operating at different points in the overall value. The implementation is complex. A stream has a descriptor with an attached list of extant values. Descriptor components are

Pointer to producer thread	# of living consumer threads	Pointers to first and last blocked consumer threads	
Pointers to first and last extant elements	# extant elements	EOS Signal	Link to next descriptor in concatenation

The attached elements each require a reference count since all consumers must see all elements. The cells for all stream elements in a Sisal program come from a special pool, not from general-purpose dynamic storage. All element types can be accommodated because nonscalar values are pointers to dynamic storage throughout the implementation.

IF2UP analyzes a program using streams to decide if a stream has a single consumer and is not a concatenation of substreams. Under these conditions (which are almost always true) we use a much simpler structure—a fixed-size circular buffer. No reference counts are needed, and synchronization is simple and minimal. As expected, this organization yields performance superior to that of the general version.

5. PERFORMANCE RESULTS

To evaluate the current status of Sisal, we compared the execution performance of equivalent Sisal and Fortran versions of the Livermore Loops and four large scientific programs on a Sequent Balance 21000. The four scientific programs were Gauss–Jordan elimination with full pivoting, RICARD, SIMPLE, and an instance of parallel simulated annealing. Four of the five Sisal codes ran as fast as the equivalent Fortran codes on one processor, and all five Sisal codes achieved good speedup. Note that we did not have to rewrite or recompile the Sisal codes to run on multiple processors; we simply increased the number of participating workers at run time. IF2MEM preallocated all arrays and built all but five in place. IF2UP eliminated all absolute copy operations, marked 29 copy operations for run time check, and eliminated approximately 97% of the reference count operations.

Table IV shows the performance results for the 24 Livermore Loops. Column 2 indicates whether the Sisal loop was parallel or sequential, and columns 3–5 give the execution performance in KFlops for the Fortran loops on one processor and the Sisal loops on one and five processors. Table V lists the execution times of the other four applications. Table VI gives compilation statistics for each program. Columns 2–4 give the number of static arrays built, preallocated, and built in-place; columns 5 and 6 list the number of copy and reference count operations before optimization; and columns 7–10 list the number of copy, conditional copy, and reference count operations after optimization and the number of artificial dependency edges introduced by IF2UP.

TABLE IV

Execution Performance of the Livermore Loops (in kflops)

Loop	P/S	Fortran	Sisal (1)	Sisal (5)	Speedup
1	P	70	77	331	4.3
2	S	58	59		
3	P	54	70	279	4.0
4	P	42	41		Too little parallel work
5	S	49	50		
6	P	50	69		Too little parallel work
7	P	88	83	393	4.7
8	P	36	55	194	3.5
9	P	85	74	255	3.4
10	P	45	25	73	2.9
11	S	37	45		
12	P	37	34	131	3.8
13	S	12	15		
14	P	28	37	94	2.5
15	P	59	54	244	4.5
16	P	75	30	89	3.0
17	S	53	46		
18	P	77	60	241	4.0
19	S	45	51		
20	S	86	90		
21	P	56	55	240	4.3
22	P	46	44	173	3.8
23	S	74	66		
24	P	50	28	101	3.6

5.1. The Livermore Loops

The Livermore Loops [30] are a set of 24 scientific kernels from production codes run at Lawrence Livermore National Laboratory. For many years scientists have used the Loops to benchmark high-performance computers. The speed of a parallel computer is a function of the system's hardware, communication topology, operating system, compilers, and the computational nature of the test suite. The Livermore Loops encompass a variety of computational structures, including independent parallel processes, recurrent processes, wavefronts, and pipelines [17]. As such, the Loops are an appropriate benchmark suite for parallel computers.

We ran the Fortran Loops without change, but we wrote the Sisal to reflect the computational nature of each loop (see [16] for an early version of the Sisal Loops). We wrote parallel algorithms unless input size was too small to justify parallel execution, or the parallel algorithm increased the number of computations to an extent not warranted by the input size or the hardware parameters of the Sequent Balance. We converted operations from column order to row order to compensate for the lack of true rectangular arrays in Sisal. For accurate measurement of both the Sisal and the Fortran codes, we executed each loop 300 times, except for Loop 4, which was so thin that we executed it 4000 times. On one processor, the harmonic mean of the Fortran and Sisal versions of the Loops were 45 and 44 KFlops, respectively.

5.1.1. *The Sequential Loops*

Discrepancies in the execution times of the Sisal and Fortran version of the sequential loops were due primarily to common subexpression removal, loop invariant removal, register allocation, paging, and the storage of multidimensional arrays in Sisal as arrays of arrays. The latter prevents

fast column access and increases the cost of array allocation and deallocation.

For Loops 4 and 6 we wrote Sisal routines that were maximally parallel, but still, IF2PART instructed the code generator not to slice the **for** expressions—the overhead of loop slicing could not be recovered by the parallel execution of the loop bodies. Loops 2 and 23 are recursive array definitions. As we explained before, Sisal does not permit such expressions. Although it is possible to write Sisal versions of both loops that do demonstrate some parallelism (see [16]), we did not do so for three reasons. First, the parallel algorithms are not natural. Second, the amount of parallel work is small—in Loop 23 the maximum number of concurrent operations is seven. Third, given a nonstrict implementation, both the parallel and the sequential versions would realize maximum parallelism.

We wrote two versions of Loops 5, 11, and 19: a sequential version and a version based on the method of recursive doubling [23, 24]. Although the latter introduces some parallelism, it increases the number of computations from $O(n)$ to $O(n \log n)$. In trial runs, the recursive doubling codes ran much slower than the sequential codes, regardless of the number of participating processors. However, they did achieve good speedup. Sisal's implementation of recursive doubling requires array concatenations and subarray selections. The compiler was able to preallocate memory for the former, but was not able to build all sections of the arrays in place (thus introducing copying). We are not sure whether the degradation in execution times resulted from the copying or the extra computations; however, it is our general impression that recursive doubling on medium-grain and coarse-grain shared-memory multiprocessors is not an appropriate technique.

5.1.2. *The Parallel Loops*

Despite incurring the overhead of parallel constructs, the Sisal implementations of the parallel loops (with the exception of Loops 10, 16, and 24) produced kiloflop rates equivalent to or better than Fortran on one processor. The discrepancies in performance were primarily the result of common subexpression elimination, loop invariant removal, register allocation, and paging. Loop 10 is an extreme example of the effects of register allocation on performance.

TABLE V

Execution Times for Four Large Scientific Programs

Program	Fortran	Sisal (1)	Sisal (processors)	Speedup
GJ	54.0 s	54.5 s	8.8 s (10)	6.2
RICARD	30.63 h	31.00 h	3.45 h (10)	9.0
SIMPLE	3081.3 s	3099.3 s	422.0 s (10)	7.3
PSA	476.6 s	956.2 s	267.8 s (5)	3.6

On a single processor, the Sisal version of Loop 16 executed 60% slower than that of Fortran. This Loop searches for a particle in a two-dimensional grid of zones subdivided into groups. The Fortran code sequentially searches each group, one at a time, and quits as soon as it finds the particle. The Sisal version examines all the groups in parallel. Since Sisal does not support asynchronous broadcasts, the processor that finds the particle cannot broadcast the discovery and stop the other processors. Consequently, the Sisal code searches the entire space. The lack of asynchronous broadcasts is a characteristic of determinate languages. Another reason for the poor performance is that the Fortran code includes a large number of *arithmetic ifs*. Each **if** statement compiles to a single comparison and a jump on a condition bit. The equivalent Sisal expression

```
if ... elseif ... else ... end if
```

compiles to two comparisons and two jump instructions.

On one processor, the Sisal implementation of Loop 24 executed 44% slower than that of Fortran. This Loop returns the first location of the minimum value in an array. The Fortran and Sisal codes are, respectively,

```
      loc := 1
      do 24 k = 2, n
24       if (X(k) .lt. X(loc)) loc = k
```

and

```
let
  min := for i in 1, n
         returns value of least X[i]
         end for;
in  for i in 1, n
    returns value of least i when X[i] = min
    end for
end for
```

The Fortran code executes only $(n - 1)$ comparisons, but the Sisal algorithm executes $3n$ comparisons. Sisal's limited repertoire of reduction operations and lack of user-defined reductions prevented use of a single expression.

Eight of the sixteen parallel loops achieved speedups of 3.8 or better. Loops 9, 10, 16, and 24 achieved smaller speedups because of insufficient parallel work. Loop 14 comprises two loops, one parallel and one sequential. The parallel loop showed good speedup, but the sequential loop amortized the gains. Despite considerable parallel work Loop 8 achieved a speedup of only 3.5. We observed that the loop spent considerable time building and recycling arrays, which idled processors. Loop 8 manipulates three-dimensional arrays which are built and recycled one dimension at a time. Although the memory subsystem can handle simultaneous

TABLE VI
Compilation Statistics

	Arrays			Before Opt		After Opt			
Programs	Built	PreA	In	Copy	RefC	Copy	Copy	RefC	ADE
Loops	76	76	76	39	1565	0	0	43	114
GJ	7	7	7	5	118	0	0	1	9
RICARD	29	29	28	17	207	0	6	7	5
SIMPLE	261	261	261	214	2066	0	19	61	347
PSA	46	46	42	18	696	0	4	41	168

requests, some sections require atomic access to shared data limiting the loop's potential parallelism. We saw the same effect, but to a smaller degree, in Loops 15 and 18, which manipulate two-dimensional arrays.

5.2. The Other Applications

5.2.1. *Gauss–Jordan Elimination*

Gauss–Jordan elimination with full pivoting solves a set of linear equations of the form

$$Ax = B,$$

where A is an $n \times n$ matrix and x and B are $n \times 1$ column vectors. The algorithm comprises n iterative steps. At each step, the largest element in a previously unselected row is found and moved onto the major diagonal. Say the element is found at position (i, j); then the element is moved onto the diagonal by interchanging rows i and j. In the new matrix, row j is the pivot row and $A(j, j)$ is the pivot element. After the interchange, A and B are reduced by the pivot row. The reduction is a parallel operation of $O(n^2)$.

IF2MEM preallocated and built all arrays in place. IF2UP eliminated all copy operations and all but one reference count operation. For $n = 100$, the execution times of the Sisal and Fortran versions on one processor were equivalent. On 10 processors, the Sisal code achieved a speedup of 6.2. Although both phases of a step (finding the pivot element and reducing the matrix) are parallel, neither phase is computationally intensive. In our implementation sequential work accounted for 6% of the execution time, which is enough to limit speedup on 10 processors to at most 6.4.

5.2.2. *RICARD*

RICARD [11] simulates experimentally observed elution patterns of proteins and ligands in a column of gel by numerical solution of a set of simultaneous second-order partial differential continuity equations. As the system evolves over time, the protein concentrations at the bottom of the column are sampled to construct the elution patterns. At each time step, the program calculates the change in protein concentrations at each level of the column due to, first, chromatography and, then, chemical reaction. The new values serve as the initial conditions for the next time step. The computations during the chromatography step are data independent, whereas the computations of the chemical reaction phase are independent across levels and dependent across proteins. Since the independent tasks are computationally intensive, the program should achieve near linear speedup on medium- and course-grain machines.

Osc preallocated memory for all the arrays, and built all but one of the arrays in place. The one array not built in place was constructed during program initialization, thus the copying was inconsequential. IF2UP eliminated all absolute copy operations, marked six copy operations for run time check, and eliminated 97% of the reference count operations. The six conditional copy operations were introduced because of *row sharing*. In the current Sisal implementation, arrays may share common rows. When the shared rows are updated, they have to be copied, but once copied, the rows are unique and can be updated in place. In RICARD, the conditional copies executed only once each. For a 1315-level, five-protein problem, the execution times of the Sisal and Fortran programs on 1 processor differed by less than 2%. On 10 processors, the Sisal program achieved a speedup of 9.0.

5.2.3. *SIMPLE*

SIMPLE [13] is a two-dimensional Lagrangian hydrodynamics code developed at Lawrence Livermore National Laboratory that simulates the behavior of a fluid in a sphere. The hydrodynamic and heat conduction equations are solved by finite difference methods. A tabular ideal gas equation for determining the relation between state variables is provided. The implementation of SIMPLE in Sisal 1.2 is straightforward and exposes considerable parallel work.

IF2MEM preallocated and built all arrays in place (261 of them). IF2UP eliminated all absolute copy operations, marked 19 copy operations for run time check, and eliminated 2005 out of 2066 reference count operations. The 19 conditional copy operations were introduced because of row sharing. They executed only once each. For 62 iterations of

a 100 × 100 grid problem, the Sisal and Fortran version of SIMPLE on 1 processor executed in 3099.3 and 3081.3 s, respectively. On 10 processors, the Sisal code realized a speedup of 7.3. Although the speedup of the Sisal code is good, it could be better. We are losing at least an equivalent of 1.5 processors in the allocation and deallocation of two-dimensional arrays. We noted the same phenomenon in some of the Livermore Loops that handled two- and three-dimensional arrays.

5.2.4. *Parallel Simulated Annealing*

Simulated annealing is a generic Monte Carlo optimization technique that has proven effective at solving many difficult combinatorial problems. In this study, we employed the method to solve the school timetable problem [1]. The objective is to assign a set of tuples to a fixed set of time slots (periods) such that no critical resource is scheduled more than once in any period. Each tuple is a record of four fields: class, room, subject, and teacher. Classes, rooms, and teachers are critical resources and subjects are not. At each step of the procedure, a tuple is chosen at random and moved to another period. If the new schedule has equivalent or lower cost, the move is accepted. If the new schedule has higher cost, the move is accepted with probability,

$$e^{(-\Delta C/T)},$$

where ΔC is the change in cost and T is a control parameter. If the move is not accepted, the tuple is returned to its original period. We parallelized the procedure by simultaneously choosing one tuple from each nonempty period and applying the move criterion to each. We then carried out the accepted moves one at a time. Note that more than one move may involve the same period.

IF2MEM preallocated memory for all the arrays and built all but 4 of the arrays in place. IF2UP removed all absolute copy operations, marked 4 copy operations for run time check, and removed all but 41 reference count operations. The 4 conditional copy operations were introduced because of the possibility of row sharing. In fact, there was no row sharing and no copying. The 4 arrays not built in place result from the expressions that add a tuple to a period. Since the old period is created on the previous iteration, the new period cannot be built in place. Although the compiler did not mark the new periods for build in place, the periods were rarely copied. This is because the Sisal run time system decouples the physical and logical sizes of arrays. If an element is removed from the high end of an array, the array's logical size shrinks by one (assuming the array can be shrunk in place), but its physical size remains constant; that is, the physical space is not released. When an element is added to the high end of an array, the run time system checks to see if there is space. If there is space, the element is added; if there is not space, the run time system allocates a new, larger space and copies the array. Whenever the run time system allocates new space, it always allocates a few extra bytes to accommodate future growth. In the school timetable problem, the periods are continually growing and shrinking as tuples are removed and added. Our implementation of arrays saved over 15,000 copies at the cost of a few hundred bytes of storage.

For a problem size of (30 periods, 300 tuples, 10 classes, 10 rooms, 10 teachers), the Sisal program ran twice as slow as the Fortran program (956.2 s versus 476.6 s). The difference is due to the allocation and deallocation of data structures in the Sisal program on every iteration. However, it is a simple optimization (loop invariant removal) to save the structures and pass them to the next iteration. We expect that once this optimization is implemented, the Sisal and Fortran execution times will be comparable. The Sisal version did achieve a speedup of 3.6 on five processors. This is quite good given the fact that the update of the schedule is sequential.

6. FUTURE PLANS

In the next few years, we plan to

1. define and implement Sisal 2.0,
2. design SisalCity, a comprehensive programming environment, and
3. develop run time systems for conventional, distributed-memory multiprocessors.

Early in 1990 we expect to release Sisal 2.0, the first revision of the language since 1985. Over the past 5 years, we have gained much experience in implementing and using Sisal 1.2. In workshop after workshop, the applicative programming model has been proven effective. After a week students with little or no knowledge of applicative languages and with no knowledge of Sisal have designed, written, debugged, and run programs of more than 100 lines. However, we can still make Sisal easier to use, more expressive, and faster without compromising our objectives. Currently, the language lacks features found in other functional programming languages and has constructs that are clumsy or severely impact performance.

New features will include higher-order functions, user-defined reductions, parameterized data types, foreign language modules, and rectangular arrays. Higher-order functions and user-defined reductions will allow users to create functions and reduction operations tailored to their exact needs. In Loop 24, we saw the effects of not having user-defined reductions. We believe we can implement a flexible, but robust, interface to modules written in foreign languages [36]. This will give us access to existing mathematics and graphics libraries, an important advance in supporting scientific computations. Sisal 2.0 will support true multidimensional arrays stored in contiguous space. Implementing multidimensional arrays as arrays of arrays was our greatest

single mistake. While we have found occasional use for "ragged" arrays (for example, as aggregates of dynamic sets), their disadvantages greatly outweigh their advantages: they prevent vectorization (constant stride exists between elements only in the last dimension), and deallocation requires complete traversal to decrement reference counts and recycle each component separately. Sisal 2.0 will include more extensive array operations such as vector operations, nonrectangular subarray selection, and a general array constructor that allows a set of expressions to contribute in parallel to parts of an array value. The physical space of a multidimensional array will contain only elements, so more efficient storage management will be possible.

Currently, we are designing a comprehensive programming environment for Sisal 2.0 based on X11 windows, called SisalCity. We will include tools to design, debug, and interpret Sisal 2.0 programs. An advantage of determinate functional programs is that if they run correctly on a uniprocessor, they are guaranteed to run correctly on any system regardless of resources or configuration. The environment will support a robust simulation package capable of simulating the logical performance of Sisal 2.0 programs on a variety of parallel architectures. In order to study the mapping problems, we will include different scheduling and partitioning heuristics. We will also design tools to collect and analyze actual performance data for certain target machines.

The run time system developed for Sisal 1.2 showed that conventional, shared-memory multiprocessors can support dataflow languages effectively. We plan to extend the run time system for conventional, distributed-memory machines with both local and global address spaces. We expect such machines to play an increasingly important role at the national laboratories. Note that our present system should port easily to the latter, providing us an important benchmark. Critical to our efforts will be efficient heuristics that map dataflow graphs (both tasks and data) to the resources of distributed machines.

ACKNOWLEDGMENTS

In a project of this size it is not possible to thank everyone; our apologizes to anyone we overlook. We acknowledge our collaborators, in particular, the groups at University of Manchester, Royal Melbourne Institute of Technology, Adelaide University, University of Southern California, Carnegie Mellon University, ETH Zurich, McGill University, and Syracuse University. Without their support and independent research, Sisal would not enjoy the large, worldwide user community it does today. Special thanks to Steve Skedzielewski, who led the Sisal research effort at Lawrence Livermore National Laboratory, Jim McGraw and Jon Ranelletti for their research contributions and support, and Don Austin of the Office of Energy Research (U.S. Department of Energy). We thank members of the Sisal research staff at Lawrence Livermore National Laboratory (Rea Simpson, Kim Yates, C. C. Lee, Patrick Miller, and David Zimmerman) and at Colorado State University (Tom Hanson, Tam Richert, and Seetharaman Harikrishnan).

This project was supported (in part) by the Office of Energy Research (U.S. Department of Energy) under Contract W-7405-Eng-48 to Lawrence Livermore National Laboratory and by the U.S. Army Research Office under Contract DAAL03-86-K-0101 to Colorado State University.

REFERENCES

1. Abramson, D. Using simulated annealing to solve school timetables: Serial and parallel algorithms. RMIT Tech. Rep. TR-112-069R, Royal Melbourne Institute of Technology, Melbourne, Australia, 1988.
2. Abramson, D., and Egan, G. K. An overview of the RMIT/CSIRO parallel system architecture project. *Austral. Comput. J.* **20,** 3 (Aug. 1988).
3. Accetta, M., *et al.* Mach: A new kernel foundation of Unix development. *Proc. USENIX 1986 Summer Conference.* USENIX, Atlanta, GA, 1986, pp. 93–112.
4. Ackerman, W. B., and Dennis, J. B. VAL—A value-oriented algorithmic language. MIT Tech. Rep. LCS/TR-218, MIT, Cambridge, MA, June 1979.
5. Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.
6. Barth, J. M. An interprocedural dataflow analysis program. *Proc. 4th ACM Symposium of the Principles of Programming Languages,* ACM, 1978, pp. 119–131.
7. Bershad, E. D., and Levy, H. M. PRESTO: A system for object-oriented parallel programming. University of Washington Tech. Rep. 87-09-01, University of Washington, Seattle, WA, Jan. 1987.
8. Cann, D. C., and Oldehoeft, R. R. Reference count and copy elimination for parallel applicative computing. Colorado State University Tech. Rep. CS-88-129, Colorado State University, Fort Collins, CO, Nov. 1988.
9. Cann, D. C., and Oldehoeft, R. R. High performance parallel applicative computing. Colorado State University Tech. Rep. CS-89-104, Colorado State University, Fort Collins, CO, Feb. 1989.
10. Cann, D. C. Compilation techniques for high performance applicative computation. Ph.D. thesis, Department of Computer Science, Colorado State University, 1989.
11. Cann, J. R., *et al.* Small zone gel chromotography of interacting systems: Theoretical and experimental evaluation of elution profiles for kinetically controlled macromolecule–ligand reactions. *Anal. Biochem.* **175,** 2 (Dec. 1988), 462–473.
12. Cohen, J. Garbage collection of linked data structures. *ACM Comput. Surveys* **13,** 3 (Sept. 1981), 341–367.
13. Crowley, W. P., Hendrickson, C. P., and Rudy, T. E. The SIMPLE code. Lawrence Livermore National Laboratory Tech. Rep. UCID-17715, Lawrence Livermore National Laboratory, Livermore, CA, Feb. 1978.
14. Denning, P. J. The working set model for program behavior. *Comm. Appl. Math. Comput.* **11,** 5 (May 1968), 323–333.
15. Dennis, J. B. Mapping programs for data parallel execution on the Connection Machine, in preparation.
16. Feo, J. T. The Livermore Loops in Sisal. Lawrence Livermore National Laboratory Tech. Rep. UCID-21159, Lawrence Livermore National Laboratory, Livermore, CA, Aug. 1987.
17. Feo, J. T. An analysis of the computational and parallel complexity of the Livermore Loops. *Parallel Comput.* **8,** 7 (July 1988), 163–185.
18. Ferrante, J., Ottenstein, K. J., and Warren, J. D. The program dependence graph and its use in optimization. Michigan Technological University Tech. Rep. CS-TR-86-8, Michigan Technological University, Houghton, MI, Aug. 1986.
19. Gross, T., and Sussman, A. Mapping a single-assignment language onto the Warp systolic array. In Kahn, G. (Ed.). *Proc. Functional Programming Languages and Computer Architecture.* Springer-Verlag, Portland, OR, 1987, pp. 347–363.
20. Gurd, J. R., Kirkham, C. C., and Watson, I. The Manchester prototype dataflow computer. *Comm. Appl. Math. Comput.* **28,** 1 (Jan. 1985), 34–52.

21. Hudak, P., and Bloss, A. The aggregate update problem in functional programming systems. *Proc. 12th ACM Symposium on the Principles of Programming Languages.* ACM, New Orleans, LA, Jan. 1985, pp. 300–313.

22. Knuth, D. *The Art of Computer Programming: Fundamental Algorithms.* Addison-Wesley, Reading, MA, 1973, Vol. 1.

23. Kogge, H. S., and Stone, P. M. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Comput.* **C-22,** 8 (Aug. 1973), 786–793.

24. Kogge, H. S. Parallel solution of recurrence problems. *IBM. J. Res. Develop.* **19,** 2 (Mar. 1975), 138–148.

25. Kuck, D. J., *et al.* Dependence graphs and compiler optimizations. *Proc. 8th ACM Symposium on the Principles of Programming Languages.* ACM, Williamsburg, VA, Jan. 1981, pp. 207–218.

26. Lee, C-C., Skedzielewski, S. K., and Feo, J. T. On the implementation of applicative languages on shared-memory, MIMD multiprocessors. *Proc. Parallel Programming: Environments, Applications, Language, and Systems Conference.* IEEE Computer Society, New Haven, CT, July 1988, pp. 188–197.

27. Lee, C-C. Experience of implementing applicative parallelism on Cray X/MP. *Proc. CONPAR '88.* British Computer Society, Manchester, England, Sept. 1988, pp. 19–25.

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

29. 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, Mar. 1985.

30. McMahon, F. H. Livermore fortran kernels: A computer test of the numerical performance range. Lawrence Livermore National Laboratory Tech. Rep. UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, Dec. 1986.

31. Nikhil, R. S. *ID Reference Manual, Version 88.1,* Computation Structures Group Memo 284, Laboratory for Computer Science, MIT, Cambridge, MA, Aug. 1988.

32. Mitrovic, S. Personal communications.

33. Oldehoeft, R. R., and Allen, S. J. Execution support for HEP Sisal. In Kowalik, J. (Ed.). *Parallel MIMD Computation: The HEP Supercomputer and Its Applications.* MIT Press, Cambridge, MA, 1985, pp. 151–180.

34. Oldehoeft, R. R., and Allen, S. J. Adaptive exact-fit storage management. *Comm. Appl. Math. Comput.* **28,** 5 (May 1985), 506–511.

35. Oldehoeft, R. R., Cann, D. C., and Allen, S. J. Sisal: Initial MIMD performance results. *Proc. 1986 Conference on Algorithms and Hardware for Parallel Processing.* Aachen, Federal Republic of Germany, Sept. 1986, pp. 120–127.

36. Oldehoeft, R. R., and McGraw, J. R. Mixed applicative and imperative programs. Lawrence Livermore National Laboratory Tech. Rep. UCRL-96244, Lawrence Livermore National Laboratory, Livermore, CA, Feb. 1987.

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

38. Padua, D., Kuck, D., and Lawrie, D. High-speed multiprocessors and compilation techniques. *IEEE Trans. Comput.* **C-29,** 9 (Sept. 1980), 763–776.

39. Padua, D., and Wolfe, M. Advanced compiler optimizations for supercomputers. *Comm. Appl. Math. Comput.* **29,** 12 (Dec. 1986), 1184–1201.

40. Polychronopoulos, C. D., and Kuck, D. J. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.* **C-36,** 12 (Dec. 1987), 1425–1439.

41. 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.

42. Sarkar, V., and Hennessey, J. Compile-time partitioning and scheduling of parallel programs. *Proc. SIGPLAN 1986 Symposium on Compiler Construction.* ACM, Palo Alto, CA, June 1986, pp. 17–26.

43. Skedzielewski, S. K., and Glauert, J. *IF1—An Intermediate Form for Applicative Languages, Lawrence Livermore National Laboratory Manual M-170.* Lawrence Livermore National Laboratory, Livermore, CA, July 1985.

44. Skedzielewski, S. K., and Yates, R. K. *Fibre: An External Format for Sisal and IF1 Data Objects, Version 1.0, Lawrence Livermore National Laboratory Manual M-154.* Lawrence Livermore National Laboratory, Livermore, CA, Jan. 1985.

45. Skedzielewski, S. K., and Welcome, M. L. Dataflow graph optimization in IF1. In Jouannaud, J. P. (Ed.). *Functional Programming Languages and Computer Architectures.* Springer-Verlag, New York, 1985, pp. 17–34.

46. Skedzielewski, S. K., and Simpson, R. J. 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.

47. Welcome, M. L., *et al. IF2: An Applicative Language Intermediate Form with Explicit Memory Management, Lawrence Livermore National Laboratory Manual M-195.* Lawrence Livermore National Laboratory, Livermore, CA, November 1986.

48. Wolfe, M. Automatic parallelism detection: What went wrong? *Proc. SRC Parallelism Packaging Workshop.* Supercomputing Research Center, Leesburg, VA, Apr. 1988.

JOHN T. FEO earned a B.A. in mathematics, physics, and astronomy at the University of Pennsylvania. He received an M.A. in astronomy and a Ph.D. in computer science at The University of Texas at Austin. He is currently the Acting Group Leader of the Computer Research Group at Lawrence Livermore National Laboratory. His research interests include parallel processing, applicative and functional programming, algorithms, and performance. Dr. Feo is a lecturer at the University of California, Davis/Livermore, and a member of the UCD Computer Science Executive Committee.

DAVID C. CANN earned a B.S. in general biology and an M.S. and a Ph.D. in computer science at Colorado State University. He is currently a member of the Computing Research Group at Lawrence Livermore National Laboratory. His research interests include parallel processing, applicative and functional programming, compilers, and operating systems. Dr. Cann is a member of the ACM and the Computer Society of the IEEE.

RODNEY R. OLDEHOEFT earned a B.S. in mathematics at Southern Illinois University and an M.S. and a Ph.D. in computer science at Purdue. He is now a professor of computer science and the department chair at Colorado State University. His research interests include parallel processing software and systems, applicative and functional programming, operating systems, and performance evaluation. Dr. Oldehoeft is a cofounder of the Sisal Project and has contributed primarily to language design, run time parallelism management, and performance optimizations. He is on the editorial board for the Wiley Series on Parallel Computing and is an accreditation visitor for the ACM/IEEE Computer Science Accreditation Board.

Received December 1, 1989; revised May 30, 1990; accepted July 31, 1990