UCRL-102263 Rev. 1
PREPRINT

SISAL 1.2: An Alternative to FORTRAN for Shared Memory Multiprocessors

David Cann
and
John Feo

Lawrence Livermore National Laboratory
Livermore, CA

This paper was prepared for the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation

White Plains, NY June 20-22, 1990

November 20, 1989

Lawrence Livermore National Laboratory

SISAL 1.2: An Alternative to FORTRAN for Shared Memory Multiprocessors *

David Cann and John Feo

Abstract

The acquisition of parallel processors in the scientific community is increasing, but the difficulties of programming parallel machines persist. Two approaches have emerged: automatic parallelizing compilers for extant languages, and new languages that provide an easier-to-use and cleaner parallel programming model. Unfortunately most new languages have acquired a reputation for inefficiency because of their semantics. This paper compares the performance of SISAL 1.2, an applicative language for parallel numerical computations, and FORTRAN using the Livermore Loops. We show that applicative programs when compiled using a set of powerful yet simple optimization techniques can achieve execution speeds comparable to FORTRAN, and can effectively exploit shared memory multiprocessors.

1 Introduction

The acquisition of parallel processors in the scientific community is increasing, but the difficulties of programming parallel machines persist. Most parallel programming languages in use today thwart programmer productivity and hinder analysis. They fail to separate problem specification and implementation, fail to emphasize modular design, and inherently hide data dependencies. In response, researchers are developing new languages of both conventional and novel design [7,9] that provide an an easier-to-use and cleaner parallel programming model. One such language is SISAL 1.2, an applicative language for parallel numerical computations. Regrettably, applicative languages have acquired a reputation for inefficiency because of their single-assignment semantics.

This paper illustrates that with some simple yet powerful compilation techniques, applicative languages can compete with conventional languages on shared memory multiprocessors. To this end, we compare the execution performance of SISAL 1.2 [7] and FORTRAN on a Sequent Balance 21000[1] using the Livermore Loops [8]. The Loops are a set of 24 computational kernels found

*This work was supported (in part) by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, and by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48 to the U.S. Department of Energy.

[1]Sequent Balance is a trademark of the Sequent Computer Corporation.

frequently in large-scale scientific applications and have been used for many years to benchmark computer system performance.

In the next section we briefly highlight the attributes of applicative languages and expound their inefficiencies. Then we present an overview of how the SISAL compiler successfully eliminates these inefficiencies. Next we present the comparisons, analyze the results, draw some conclusions, and introduce future work.

2 Applicative Computation

An applicative program is a collection of function definitions and applications, where a function defines a side effect free correspondence between members of its domain and members of its range.

The merits of this simple programming model are far reaching [5,15]. First, programs are inherently modular, hence easier to write, debug, and maintain. Second, programs describe data dependence graphs; thus compilers can spend more time restructuring programs and less time unraveling their behavior. Third, programs are determinate. If they run correctly on one processor, they run correctly, without exception, on multiple processors—programmers need not debug parallel execution or understand its complexities. Without optimization, however, the overhead of applicative computation can be high. Implementations that adhere religiously to applicative semantics must copy data when deriving new values. For languages like SISAL, which support arrays, this copying can severely degrade performance and make the use of applicative languages infeasible.

Most copying results from operations that build new aggregates and operations that modify extant aggregates. Consider the SISAL **for** expression shown in Figure 1, which returns an array of 100 elements (`A`). In unoptimized form, this expression builds 99 intermediate arrays, each one element larger than the previous, and requires 100 memory allocation requests, 99 memory deallocation operations, and 4950 double precision move operations. On the other hand, our compiler preallocates an array of 100 elements and stores each element directly into memory, thus eliminating the intermediate arrays and all the associated operations. Now consider the expression `A[5: 0.0d0]`, which changes the `5`'th element of `A` to zero. Even though this is the last use of `A`, strict adherence to applicative semantics would require us to build an entirely new array. Our compiler recognizes that this is the last use of `A` and generates code to update it

```
type double = double_real;
type OneD   = array[double];

function Build( returns OneD )
   let
      A := for I in 1, 100
           returns array of
              sqrt( double_real( I ) )
           end for
   in
      A[5: 0.0d0]
   end let
end function
```

Figure 1: A SISAL function constructing an array.

in-place.

An additional source of inefficiency in SISAL 1.2, although not a product of its applicative semantics, is its representation of n-dimensional arrays as arrays of arrays. This can cause excessive storage allocation and deallocation requests, and overhead when dereferencing columns or planes.

3 The SISAL Compiler and Run Time System

In this section we present a brief overview of the SISAL compiler and run time system. For a detailed discussion see [1] and [11]. Figure 2 depicts the SISAL compilation process. First, a front end translates SISAL source into IF1 [12], an intermediate form defining data flow graphs. The compiler then forms a monolithic IF1 program (linking all separately compiled files) and runs a machine independent optimizer to expand function calls, move invariant code, eliminate common subexpressions, fuse loops, fold constants, and remove dead code [14].

Next a build-in-place analyzer inserts code to preallocate array storage where analysis or expressions executed at run time can calculate array sizes [10]. During this analysis, the compiler translates the IF1 monolith into IF2 [16]. Since IF2 includes explicit memory management operations, the compiler can now optimize these operations. Additionally, IF2 provides artificial dependence edges to constrain execution order and reference count operations to control storage

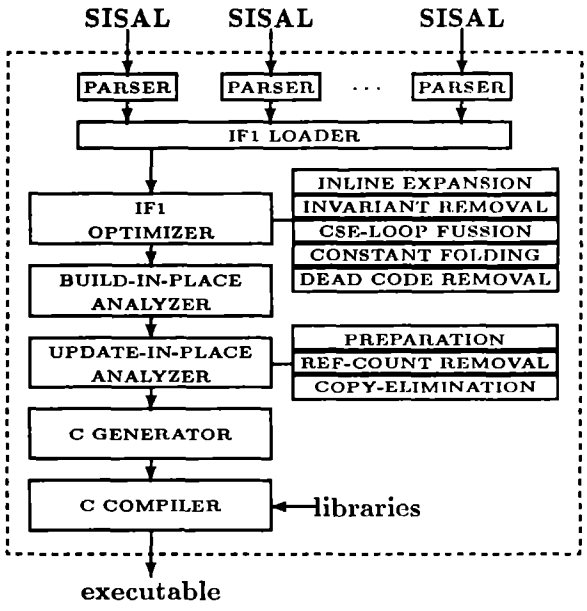

Figure 2: SISAL language processing.

reclamation.

After inserting code to preallocate memory, the compiler identifies those operations that can directly modify arguments without corrupting program semantics [1,3,4,13]. The analysis proceeds in three phases. Phase one inserts explicit copy operations to decouple copy logic from aggregate modifiers, and adds reference count operations to decouple storage management from all aggregate read and write operations. Phase two inserts artificial dependence edges to promote early execution of aggregate read operations and to delay execution of copy operations. Then it eliminates all unnecessary reference count operations. Phase three eliminates the unnecessary copy operations and tags those that require run time analysis for copy avoidance. The analysis considers iteration, handles nested aggregates, and crosses function boundaries.

Finally, the compiler translates the optimized program into C, and inserts calls to the run time library to support parallelism. We chose the C programming language as an intermediate form to expedite compiler development, increase compiler portability, and allow manual experimentation with various optimizations. Unfortunately the local C compiler can dictate final performance. For the Sequent Balance we wrote a simple machine dependent optimizer, working at the assembly language level, to improve register utilization and reduce code size.

The SISAL run time system is a microtasking kernel tuned for the parallel execution of loops [11]. After execution begins, the kernel creates and assigns a worker process to each participating

processor. The workers then spin wait for loops to appear in a global loop pool. When a loop appears, each worker grabs a slice of the loop[2], acquires a run time stack from the memory management subsystem (unless one is already owned), executes the slice, and returns to the pool. If during execution the slice must wait for completion of a storage request or the results of another loop selected for parallel execution, the governing worker will save its hardware state and record the outstanding event on the appropriate list. When the event completes, a worker will restore the slice and continue its execution. By default, the system breaks each loop into n slices, where n is the number of participating processors.

The current SISAL run time system does not spawn user functions as separate tasks; instead we expand all functions calls. We have found on medium-grain machines like the Sequent that we rarely recover the cost of a spawn, and on coarse-grain machines like the Cray-XM/P that the overhead often hurts performance [6].

4 FORTRAN versus SISAL

The Livermore Loops [8] are a set of 24 scientific kernels from production codes run at Lawrence Livermore National Laboratory. They encompass a variety of computational structures, including independent parallel processes, recurrent processes, wavefronts, and pipelines [2]. For many years scientists have used the Loops to benchmark high performance computers. Here we use the Loops to compare the execution speed of SISAL 1.2 and FORTRAN on a Sequent Balance 21000.

We ran the FORTRAN loops without change. The FORTRAN compiler provided on the Sequent folded constants, allocated registers across subroutines and basic blocks, and optimized array index computations within `DO` loops.

We wrote the SISAL to reflect the computational nature of each Loop, and did not tailor the algorithms for either the compiler or run time system. In general, if the Loop was inherently sequential, we used SISAL's **for initial** expression. If the Loop was inherently parallel, we used **for** expressions. In certain instances, however, foreknowledge of input size did influence our coding. For example, we wrote sequential implementations of Loops 2, 4, 6, and 23 because their input data sizes were too small to warrant parallel execution. In comparison to the FORTRAN codes

[2]A slice is an autonomous computational unit comprised of one or more consecutive loop iterations.

```
type double = double_real;
type OneD   = array[double];

function Loop1( n:integer; Q,R,T:double; Y,Z:OneD returns OneD )
   for K in 1,n
       X := Q + (Y[K] * (R * Z[K+10] + T * Z[K+11]))
   returns array of X
   end for
end function

function Main( rep,n:integer; Q,R,T:double; Y,Z:OneD returns OneD )
   for i in 1, rep
      X := Loop1( n, Q, R, T, Y, Z );
   returns value of X
   end for
end function
```

Figure 3: SISAL code for Livermore Loop 1.

we changed from column-order to row-order to help compensate for the lack of true rectangular arrays in SISAL and, where ever possible, maintained a similar output structure. For more accurate measurement of both the SISAL and FORTRAN codes, we executed each Loop 300 times (Loop 4 is so thin that we had to execute it 4000 times). As an example, Figure 3 gives the complete SISAL source for Loop 1.

Table 1 shows the performance results, where execution times are in kiloflops. Table 2 summarizes the data in Table 1, showing minimum and maximum kiloflop rates, and the arithmetic and harmonic means. For FORTRAN we only report single processor rates, but for SISAL we report achieved kiloflops on one and five processors. The letters P and S in Table 1 show whether the SISAL algorithm was parallel or sequential, respectively. The complexity column shows the dimensionality of the arrays referenced in each Loop. Note we did not have to recompile the SISAL codes to run on five processors; we simply increased the number of participating workers. This epitomizes the advantages of applicative programming.

For the single processor runs, 11 of the SISAL Loops ran faster than, or within 1% of FORTRAN; 6 of the SISAL Loops ran within 20% of FORTRAN; and 2 of the SISAL Loops ran within 34% of FORTRAN. The remaining 5 (Loops 8, 16, 18, 23, and 24) did not fare as well. In general, this shows that sequential SISAL and sequential FORTRAN performance is comparable.

Table 1: Kiloflop rates on the Sequent Balance for the Livermore Loops.

Loop number	array complexity	algorithm type	FORTRAN 1 processor	SISAL 1 processor	SISAL 5 processors
1	1D	P	70	76	333
2	1D	S	58	58	58
3	1D	P	54	70	281
4	1D	S	42	42	42
5	1D	S	49	49	49
6	1D	S	50	49	49
7	1D	P	88	83	395
8	3D	P	36	16	33
9	2D	P	85	74	252
10	2D	P	45	39	91
11	1D	S	37	47	47
12	1D	P	37	34	131
13	1D,2D	S	12	13	13
14	1D	P	28	44	101
15	2D	P	59	44	136
16	1D	P	75	13	38
17	1D	S	53	45	45
18	2D	P	77	29	55
19	1D	S	45	51	51
20	1D	S	86	90	90
21	2D	P	56	54	224
22	1D	P	46	45	177
23	2D	S	74	42	42
24	1D	P	50	27	101

Table 2: Summary of kiloflop rates on the Sequent Balance for the Livermore Loops.

key	FORTRAN	SISAL 1 processor	SISAL 5 processors
minimum	12	13	13
maximum	88	90	395
arithmetic mean	55	47	118
harmonic mean	45	36	60

The parallel SISAL implementations achieved an average speedup of about 3.4 on five processors. In general, the SISAL compiler eliminated 97% of the reference count operations and all the copying. For the multidimensional problems, however, the costs for referencing arrays of arrays was evident.

4.1 The Sequential Loops

Of the sequential SISAL Loops, only Loop 23 did not yield performance similar to FORTRAN. The 34% increase in execution time was the direct result of SISAL's representation of two dimensional arrays; that is, its inability to traverse columns efficiently.

Loops 2, 4, 6, and 23 have parallel implementations in SISAL, but we chose to use their sequential implementations as problem size did not justify run time overhead. The parallelism was in innermost loops. Also we chose not to use the parallel implementations of Loops 5, 11, and 19, which require recursive doubling to expose parallelism [2]. Recursive doubling is $O(Log\ n)$ in time, but requires $O(n\ Log\ n)$ computations, whereas the equivalent sequential algorithm requires $O(n)$ computations, but is $O(n)$ in time. In trial runs, the parallel SISAL implementations ran much slower than the sequential codes, regardless of the number of participating processors. However, they did achieving reasonable 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. We are not sure whether the degradation in execution times resulted from the copying or the extra computations intrinsic to recursive doubling, but it is our general impression that recursive doubling on medium-grain and coarse-grain shared memory multiprocessors is not an appropriate technique.

4.2 The Parallel Loops

Despite incurring the overhead of parallel constructs, the SISAL implementations of Loops 1, 3, 7, 9, 10, 12, 14, 21, and 22 produced kiloflop rates equivalent to, or better than FORTRAN on one processor and, except for Loop 14, showed good speedup on 5 processors. The parallel performance of Loop 14 was not the result of SISAL semantics or compiler deficiencies. Loop 14 comprises two adjacent loops, one inherently parallel and one with carried dependencies preventing parallel execution. The parallel loop showed good speedup, but the sequential loop amortized

```
type double = double_real;
type OneD   = array[double];

function Loop24( n:integer; X:OneD returns integer )
  let
      l := for y in X
           returns value of least y
           end for
  in
      for y in X at i returns
      value of least i when y = l
      end for
  end let
end function

function Main( rep,n:integer; X:OneD returns integer )
   for i in 1, rep
      v1 := Loop24( n, X );
   returns value of v1
   end for
end function
```

Figure 4: SISAL code for Livermore Loop 24.

the gains. Consequentially, five processors only doubled the kiloflop rate.

On one processor, the SISAL implementation of Loop 24 executed 80% slower than FORTRAN, but immediately overtook it on two processors and doubled its kiloflop rate on five processors. This Loop finds the location of the first minimum in an array. Figure 4 shows the SISAL implementation. The FORTRAN version only requires a single loop, but the SISAL algorithm requires two **for** expressions. SISAL's limited repertoire of reduction operations (sum, product, minimum, maximum, and catenate) and lack of user-defined reductions prevented use of a single expression. SISAL 2.0 will include user-defined reductions.

The SISAL implementation of Loop 16 is 100% parallelizable, but it could not out-perform FORTRAN. This Loop searches for a particle in a two-dimensional grid of zones subdivided into groups. The FORTRAN Loop 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, but searches the entire space because the language does not support asynchronous broadcasts—the processor finding the particle cannot broadcast the event and stop the other processors. The lack

of asynchronous broadcasts is a characteristic of determinate languages.

Loops 8, 15, and 18 did not do well, yet each is parallel and comprises considerable work. Loop 8 manipulates three-dimensional arrays, and the other two manipulate two-dimensional arrays. Using a profile facility built into the SISAL run time kernel, we observed that Loop 8 spent 46% of its time (and Loop 18 spent 18% of its time) allocating and deallocating array storage (that is, builting arrays during Loop execution and recycling them between repetitions). The profile also showed that memory requests were idling processors. Although the memory subsystem can handle simultaneous storage requests, some sections require atomic access to shared data. In general, the lack of true multidimensional arrays contributed to the timing discrepancies for all three loops. SISAL 2.0 will support true multidimensional arrays in the spirit of FORTRAN. The allocation and deallocation of such structures will be as efficient as that for one-dimensional arrays in the current implementation.

5 Conclusions

In this paper we have shown that applicative languages can compete with conventional languages, and are a viable tool for exploiting shared memory multiprocessors. The scientific community should not consider applicative languages inefficient, or ignore their potential. Given the expressive and easy-to-use parallel programming model they provide, these languages represent an attractive alternate to conventional programming languages on shared memory multiprocessors.

We are currently revising the definition of SISAL to eliminate its known deficiencies. First we are adding true rectangular arrays. The overhead of arrays of arrays is just too high, as seen in this paper. Second, to enhance expressive power, we are adding parameterized types, modules, high-order functions, and user defined reductions. We are also merging the two loop forms. We plan to implement the revised language on both shared and distributed memory multiprocessors.

Acknowledgements

We would like to thank Dr. Rod Oldehoeft, Chairman of the Computer Science Department at Colorado State University, for providing access to the Department's Sequent, and for his contributions to SISAL's design and implementation.

References

[1] D. C. Cann. *Compilation Techniques for High Performance Applicative Computation.* PhD thesis, Colorado State University, Computer Science Department, Fort Collins, CO, 1989.

[2] J. T. Feo. An analysis of the computational and parallel complexity of the Livermore Loops. *Parallel Computing*, 1988. To appear.

[3] P. Hudak. A semantic model of reference counting and its abstraction. In *Proceedings of the ACM Conference on Lisp and functional programming*, pages 351–363, August 1986.

[4] P. Hudak and A. Bloss. The aggregate update problem in functional programming systems. In *Twelfth Annual ACM Conference of the Principles of Programming Languages*, pages 300–313, January 1985.

[5] J. Hughes. Why functional programming matters. Technical Report PMG-40, Chalmers Tekniska Hogskola, 1984.

[6] C. Lee. Experience of implementing applicative parallelism on Cray X-MP. Technical Report UCRL-98303, Lawrence Livermore National Laboratory, May 1988.

[7] J. R. McGraw, S. K. Skedzielewski, S. J. Allan, R. R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce, and R. Thomas. SISAL: Streams and iteration in a single assignment language: Reference manual version 1.2. Manual M-146, Rev. 1, Lawrence Livermore National Laboratory, Livermore, CA, March 1985.

[8] F. H. McMahon. The Livermore Fortran kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.

[9] R. Nikhil. Id Nouveau: Quick reference guide. Technical report, MIT Laboratory for Computer Science, Cambridge, MA, January 1987.

[10] J. E. Ranelletti. *Graph Transformation Algorithms for Array Memory Optimization in Applicative Languages.* PhD thesis, University of California at Davis, Computer Science Department, Davis, California, 1987.

[11] T. R. Richert. Efficient task management for SISAL. Technical Report 89-111, Computer Science Department, Colorado State University, July 1989.

[12] S. Skedzielewski and J. Glauert. IF1—an intermediate form for applicative languages. Manual M-170, Lawrence Livermore National Laboratory, Livermore, CA, July 1985.

[13] S. K. Skedzielewski and R. Simpson. A simple method to remove reference counting in applicative programs. Technical Report UCRL-100156, University of California Lawrence Livermore National Laboratory, November 1988.

[14] S. K. Skedzielewski and M. L. Welcome. Data flow graph optimization in IF1. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 17–34. Springer-Verlag, New York, NY, September 1985.

[15] D. A. Turner. The semantic elegance of applicative languages. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 85–92, October 1981.

[16] M. L. Welcome, S. K. Skedzielewski, R. K. Yates, and J. E. Ranelletti. IF2: an applicative language intermediate form with explicit memory management. Manual M-195, University of California Lawrence Livermore National Laboratory, November 1986.