2

CONF-901221--1

UCRL- JC-103980
PREPRINT

SISAL 1.2: High-Performance Applicative Computing

David C. Cann
John T. Feo
Thomas M. DeBoni

Lawrence Livermore National Laboratory
Livermore, CA

This paper was prepared for submittal to the 2nd IEEE Symposium on Parallel & Distributed Processing, December, 9-13, 1990, Dallas, TX

May 1990

Lawrence Livermore National Laboratory

SISAL 1.2: High-Performance Applicative Computing *

David C. Cann
John T. Feo
Thomas M. DeBoni
Computer Research Group (L-306), Lawrence Livermore Nat. Lab., P.O. Box 808, Livermore, CA 94550

UCRL-JC--103980

DE90 011461

Abstract: The acquisition of parallel processors in the scientific community is increasing, but the difficulties of parallel programming persist. Three approaches have emerged: automatic parallelizing compilers for extant languages, extended languages, and new languages that provide a cleaner and easier-to-use parallel programming model. One such new language is SISAL 1.2 [13], a general-purpose applicative language. Regrettably, applicative languages have acquired a reputation for inefficiency because of their single-assignment semantics, and dynamic creation of aggregate objects. We show that a set of powerful yet simple optimization techniques can reduce the overhead of applicative semantics without sacrificing parallelism. Optimized SISAL codes can achieve execution speeds comparable to FORTRAN, and effectively exploit shared-memory multiprocessors.

---

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

1. Introduction

The acquisition of parallel processors in the scientific community is increasing, but the difficulties of parallel programming persist. Three approaches have emerged: automatic parallelizing compilers for sequential languages, extended languages, and new languages that provide a cleaner and easier-to-use parallel programming model. Automatic parallelizing compilers have failed to meet expectation; moreover, we believe that sequential languages restrict the formulation of parallel algorithms and will always deter the automatic exploitation of parallel architectures. Programming in extended languages has proven arduous and error prone. Such languages 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 [11,13,15] that provide cleaner and easier-to-use parallel programming models.

One such language is SISAL 1.2, a general-purpose applicative language. Regrettably, applicative languages have acquired a reputation for inefficiency because of their single-assignment semantics, and dynamic creation of aggregate objects. Strict adherence to single assignment semantics requires that operations deriving new values copy their operands. In large scientific computations, such copying can significantly limit performance and consume large amounts of memory. Because of the dynamics of applicative programs, the sizes of aggregates is often not known until runtime. Allocating and deallocating memory at runtime can degrade performance and limit parallelism.

This paper illustrates that a set of simple yet powerful compilation techniques can reduce the overhead of applicative semantics without sacrificing parallelism. These optimizations are *O(n)* and, in practice, add little to the compilation times of programs. In the next section we highlight the attributes of applicative languages and discuss their inefficiencies. In section three we present an overview of the SISAL compiler and our optimization techniques. In section four we present the performance of five large scientific codes written in Sisal. In each case, our optimization techniques significantly reduced the semantic overhead of the SISAL programs without sacrificing parallelism. On a single processor, four of the five optimized SISAL programs executed as fast as the equivalent FORTRAN programs. In section five we draw some conclusions and introduce future work.

2. Applicative Computations

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. First, programs are inherently modular, hence easier to write, debug, and maintain. Second, programs describe data dependence graphs. There is a clean separation between data and control dependencies; 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 on multiple processors – there are no time dependent errors. However, without optimization, 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, that 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 or modify extant aggregates. Consider the following SISAL **for** expression, which returns an array of 100 elements

```
A := for i in 1, 100
       returns array of sqrt(double_real(i))
       end for
```

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. If the loop is sliced[1] and the slices executed concurrently, the components will be built separately. After all the slices have finished, the parent task will gather the components together, further increasing copy costs. 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. If the loop is sliced, each slice includes the starting address of its segment of the array.

Now consider the expression

```
A[5: 0.0d0]
```

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

which changes the 5th element of A to zero. Even if this were the last use of A, strict adherence to applicative semantics would require us to build an entirely new array. Our compiler recognizes most instances of last use and generates code to update in-place; that is, destructively update aggregates.

In SISAL, the size and shape of all aggregate objects are defined by the expressions that build them. This increases SISAL's expressive power, but also increases runtime overhead to allocate and deallocate memory, and to decide when deallocation should occur. A number of memory management schemes exist [5]. One scheme, regaining popularity, is reference counting. Its major advantages are: conduciveness to real-time processing, ease of implementation, and natural support for aggregate sharing and copy avoidance. However, if done improperly, reference counting can reduce sequential performance and parallel efficiency [16]. Consider an aggregate that is read by n concurrent tasks. Since each task decrements the aggregate's reference count in a critical section, the reads are sequentialized. For most programs, our compiler optimizes away over 96% of the reference count operations while reclaiming memory as soon as possible.

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 an array of $(n-1)$-dimensional arrays. This can cause excessive storage allocation and deallocation requests, and can be a source of overhead when dereferencing columns or planes.

3. The SISAL Compiler and Runtime System

In this section we present a brief overview of the SISAL compiler (Osc) and runtime system. For a detailed discussion see [3,8,17,18]. Figure 1 depicts the SISAL compilation process. First, a front end translates SISAL source into IF1 [19], an intermediate form defining data flow graphs. Second, the compiler forms a monolithic IF1 program (linking all separately compiled files) and runs a machine independent optimizer (IF1OPT) to expand function calls, move invariant code, eliminate common subexpressions, fuse loops, fold constants, and remove dead code [20].

Third, the *build-in-place* analyzer (IF2MEM) inserts explicit memory operations to preallocate array storage wherever possible [17]. During this analysis, the compiler translates the IF1 monolith into IF2 [22], an extension of IF1 that includes explicit memory management nodes. Fourth, the *update-in-place* analyzer (IF2UP) tries to minimize copy and reference count operations [2,9,10,21]. At all times preserving program correctness, this analyzer restructures some graphs to help identify operations that can execute in-place and to improve chances for in-place op-

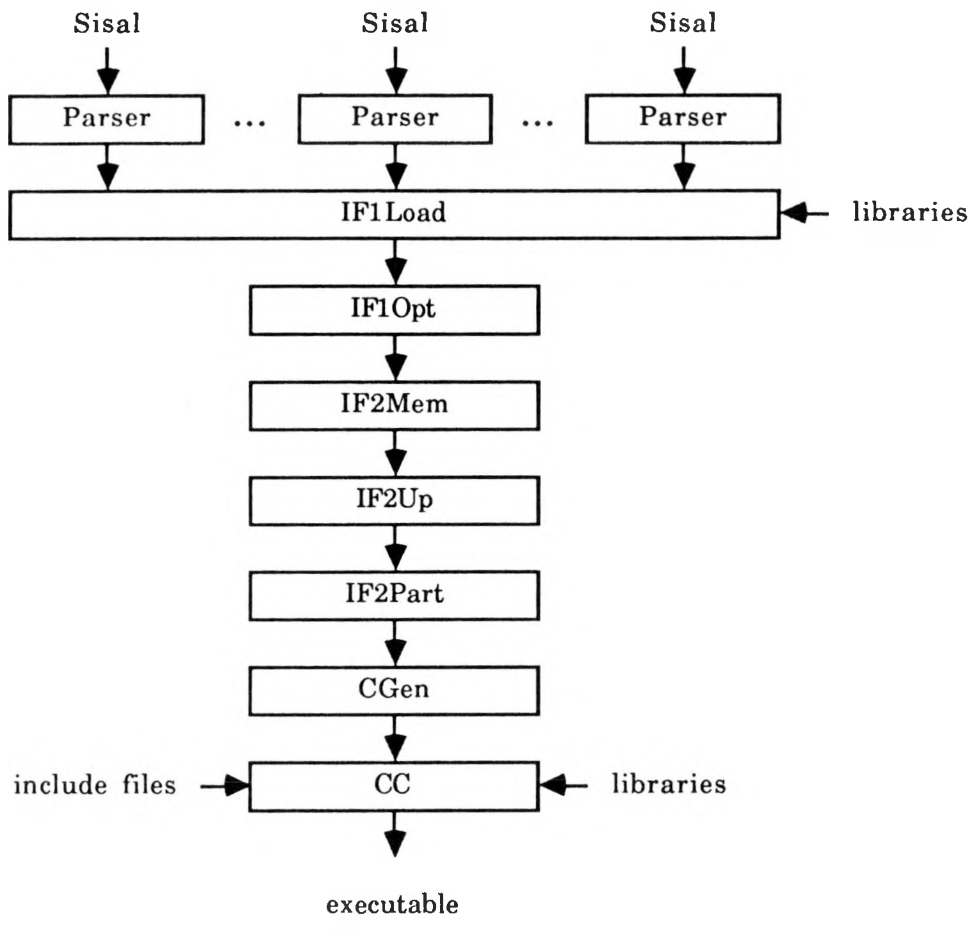


Figure 1 - Sisal language processing

eration at runtime when analysis fails. Fifth, IF2PART defines the desired granularity of parallelism based on estimates of execution time.

Finally, CGEN translates the optimized IF2 graphs into C, which is compiled into executable form using the local C compiler. Calls to the SISAL runtime system, linked during this phase of compilation, provide support for parallel execution, storage management, and I/O. We choose C as an intermediate form to shorten development time and increase portability; however, the quality of most C compilers is poor. On the Sequent Balance 21000[2], we reduced the execution times of some SISAL codes by 25% after implementing a simple machine dependent optimizer to better utilize scratch registers and reduce overall code size. We expect SISAL's performance to improve as C compilers mature.

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

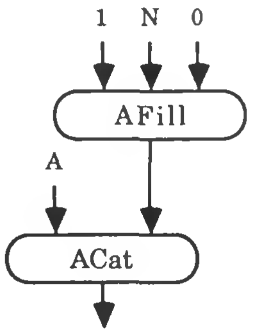
3.1 Build-in-place analysis

The build-in-place analyzer is a two pass algorithm that operates on IF1 dataflow graphs. Pass one inserts code to preallocate array storage where analysis or runtime calculations can determine array sizes. Pass two "rewires" the graph, where possible, to build array components in-place. Both passes are linear in the number of nodes in the IF1 graph. The result of this analysis is a semantically equivalent program graph in IF2. The graph now includes explicit memory management operations (called **AT-nodes**).

Consider the SISAL expression

```
B := A || array_fill(1,N,0)
```

which concatenates *A* and an array of zeros (note that *A* is an array object). Its IF1 graph is



Without optimization, the **AFill** node calls the memory manager to allocate space for an array of *N* integers, fills the space with zeros, and passes the array's address to the **ACat** node. The catenate operation then calls the memory manager to allocate memory for *B*, of size *Size(A)* + *N*, and copies into that space *A* and the array of zeros. In all, the memory manager is called twice and *Size(A)* + *N* integer values are copied.

IF2MEM creates the graph shown in Figure 2. The first three nodes along the left-hand side compute the size of *B*. The **MemAlloc** node then allocates memory for *B* (its result is B's address). The three nodes along the right-hand side preceding the **AFillAT** node compute the starting address of the array of zeros within *B*. The **AFillAT** and **ACatAT** nodes are identical to the **AFill** and **ACat** nodes in the unoptimized graph except that the **AT**-nodes are told where to build their results, whereas the non-**AT**-nodes call the memory manager to allocate space for their results. Notice that the optimizer has marked the edge from the **AFillAT** node to the **ACatAT** node with a **P**, indicating that the array of zeros is in-place. If IF2MEM also builds *A* in place, then both data edges to the **ACatAT** node will be marked with a **P**. The code generator will then replace the

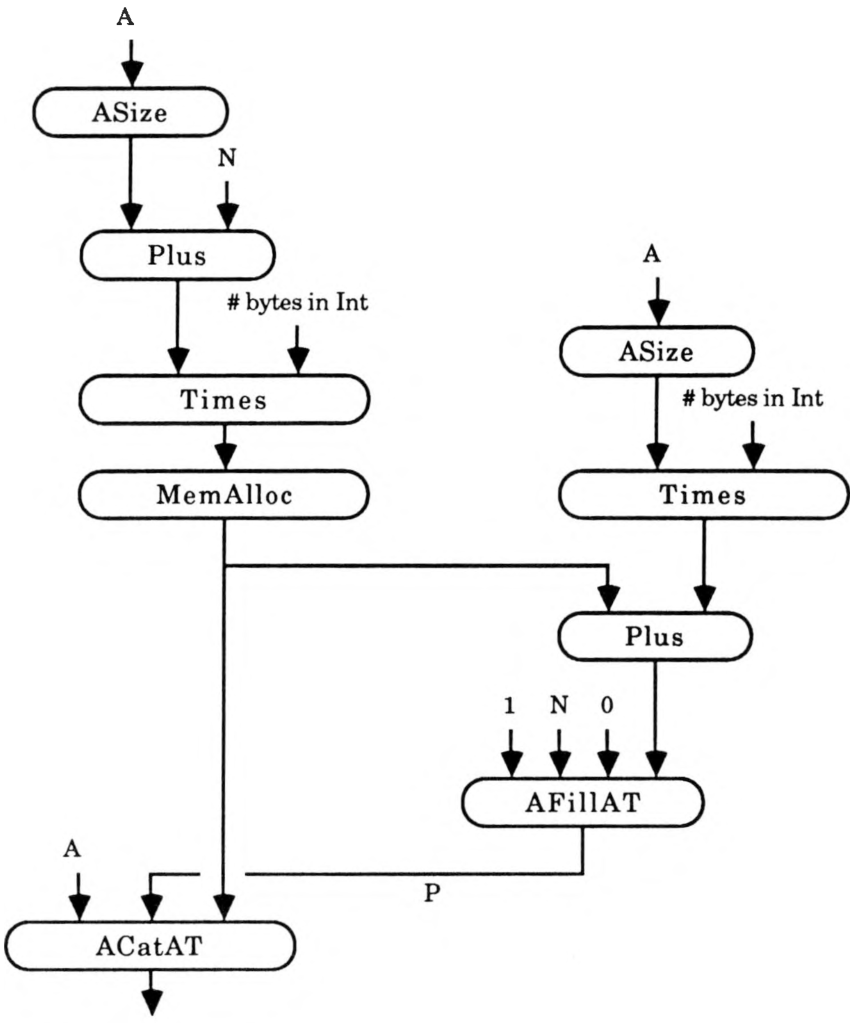


Figure 2 - Optimized IF2 graph for memory preallocation

ACatAT node with a **NoOp**, removing the catenate operation completely – overhead and all. As is, the graph in Figure 2 calls the memory manager once and copies *Size(A)* integer values.

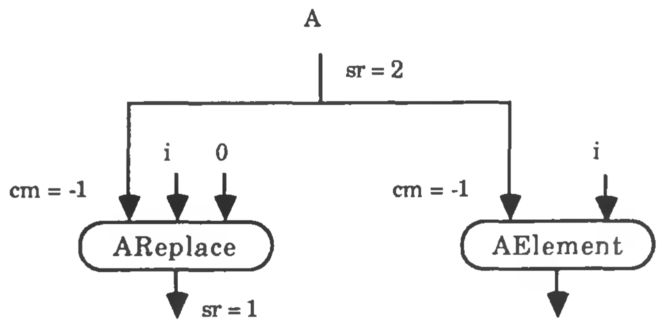
3.2 Update-in-place analysis

After inserting code to preallocate memory, the compiler rearranges nodes and introduces artificial dependencies to reduce copy and reference count operations. The analysis considers iteration, handles nested aggregates, and crosses function boundaries. It proceeds in three phases. Phase one inserts explicit copy and reference count operations. Phase two inserts artificial dependence edges to schedule read operations before write operations and eliminates all unnecessary reference count operations. Phase three eliminates the unnecessary copy operations and tags those that require runtime analysis for copy avoidance. Each phase is linear in the number of nodes in the IF1 graph.

Consider the SISAL expression

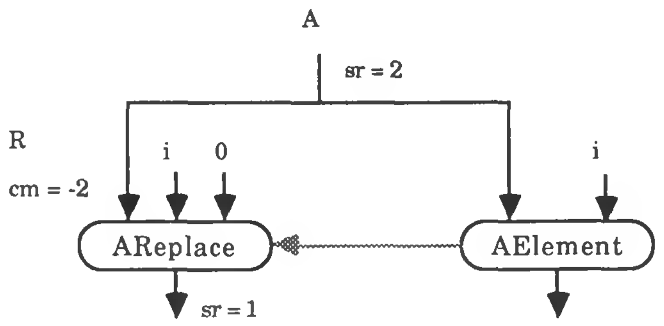
```
            B, C := A[i: 0], A[i];
```

and its IF1 graph

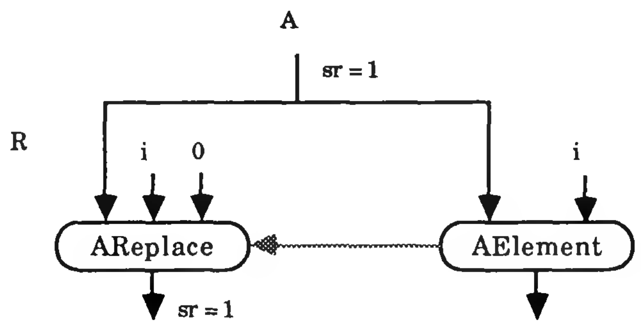


Without optimization, the runtime system must copy *A* since the read operation may execute after the update operation. **sr** and **cm** are pragmas that set and decrement the reference count of aggregate objects, respectively.

To eliminate the copy, IF2UP introduces an artificial dependence from the **AElement** node to the **AReplace** node and marks the AReplace node for in-place operation (**R**). IF2UP then pushes the reference count operation of the **AElement** node down to the **AReplace** node, changing the **cm = -1** pragma to **cm = -2,**



Since the **AReplace** node updates the array in-place and resets its reference count, the **cm** pragma is eliminated in the final graph, (note that the reference count of *A* is now set to 1).



In all, IF2UP has eliminated a copy of *A* and two reference count operations.

While the introduction of artificial dependences does reduce potential parallelism, only fine-grain dataflow machines can exploit this extra parallelism. On a medium- or course-grain machine, the read and update operations the preceding example will execute on the same processor. In this case, the artificial dependence edge does not limit actual parallelism, but simply eliminates a useless copy. In those instances in which IF2UP can not guarantee the safety of update-in-place, the decision is postponed until runtime (the update operation is marked for conditional copy, **r**). Since we usually avoid the copy, the runtime test is cost effective.

3.3 SISAL runtime system

The SISAL runtime system is a microtasking kernel that supports only stream and loop parallelism. Currently, we do not spawn user functions as separate tasks; instead we expand all functions inline. We have found that on medium-grain machines like the Sequent we rarely recover the cost of a spawn, and that on coarse-grain machines like the Cray-XM/P the overhead actually hurts performance [12]. A command line option specifies the number of worker processes (the default is 1). After execution begins, the kernel creates and assigns a worker process to each participating processor. The workers then spin-wait for producer or consumer tasks to appear on a ready list or loop tasks to appear in a global loop pool. If a task blocks, waiting for the completion of a storage request or the results of another task, the governing worker saves its hardware state, records the outstanding event, and returns to the ready state. Thus, workers never spin wait unless there is no work to do.

In SISAL, streams are non-strict by definition. To implement parallelism, stream producers and consumers are packaged by the SISAL compiler as separate tasks, and a bounded buffer is created to store stream values (the buffer's size is a runtime option). If a consumer attempts to use a value not yet produced, it blocks. Likewise, a producer will block if the buffer is full. To prevent continuous blocking and unblocking, producers and consumers awake only after some number of values have been consumed or produced, respectively. The user may set these numbers at runtime or use the default values.

When a loop appears in the global loop pool, each worker takes a slice of the loop, executes the slice, and returns to the pool for another slice. Based on estimates of execution times, IF2PART decides which loops should be sliced. A loop is either not sliced or sliced into n chunks, where n is a runtime option (the default is the number of workers).

4. Reductions in semantic overhead

In sections 2 and 3, we illustrated the high overhead of single-assignment semantics and briefly described how our compiler reduces, and even eliminates, this overhead without violating SISAL's applicative semantics. In this section, we compare the number of memory management, copy, and reference count operations before and after optimization in five scientific codes: the Livermore Loops, Gauss-Jordan elimination with full pivoting, RICARD, SIMPLE, and Parallel Simulated Annealing. Table I gives the compilation statistics for the five programs: columns 2-4 list the number of 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. All five programs achieved good speedup on the Sequent Balance 21000 and four out of the five programs ran as fast as equivalent FORTRAN programs on one processor.

4.1 The Livermore Loops

The Livermore Loops [14] 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 kernels include both sequential and parallel computations of varying complexity [7]. In general, we used iterative for expressions to implement the sequential loops, and parallel for expressions to implement the parallel loops. Because the input data sizes of Loops 2 and 23 are too small to justify parallel execution, we wrote sequential versions of each loop.

Statically, the SISAL version of the Loops builds 76 arrays. IF2MEM preallocated memory and built all the arrays in place. IF2UP eliminated all 39 copy operations, reduced the number of reference count operations from 1565 to 43, and introduced 114 artificial dependence edges. The harmonic means of the execution speeds of the SISAL Loops on one and five processors were 44 and 77

Programs	Arrays			Before Opt		After Opt			
	Built	PreA	In	Copy	RefC	Copy	Ccopy	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

Table I - Compilation Statistics

KFlops (thousands of floating point operations per second), respectively. In comparison, the harmonic mean of the FORTRAN Loops was 45 KFlops. With copy and reference count operations all but eliminated, it is not surprising that the SISAL Loops ran as fast as the FORTRAN Loops.

Eight of the sixteen parallel loops achieved speedups of 3.8 or better. Seven of the sixteen loops (Loops 2, 6, 9, 10, 14, 16, and 24) achieved smaller speedups due to insufficient parallel work. In fact, the amount of parallel work in Loops 4 and 6 was so small that the compiler generated sequential code. Only Loop 8 exhibited poor speedup. Using a profile facility built into the SISAL runtime kernel, we observed that Loop 8 spent considerable time building and recycling arrays. The profile showed that the memory operations were idling processors. Loops 8 manipulates three dimensional arrays, which SISAL 1.2 stores as arrays of arrays. The structures are built and recycled one dimension at a time. Although the memory subsystem can handle simultaneous storage requests, some sections require atomic access to shared data, limiting potential parallelism. We saw the same effect, but to a smaller degree, in some loops that manipulate two dimensional arrays (Loops 15 and 18).

4.2 Gauss-Jordan Elimination with full pivoting

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

$$A\ x = B$$

where A is an $n \times n$ matrix and x and B are $n \times 1$ column vectors. The 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 effect of the reductions are to transform A into the identity matrix and B into x.

The static SISAL program builds 7 arrays. All the arrays are preallocated and built in place. Before optimization, the code included 5 copy operations and 118 reference count operations. After optimization, the code included no copy operations, 1 reference count operation, and 9 artificial dependency edges. For $n = 100$, the optimized SISAL program ran in 54.5 seconds on one processor and in 8.8 seconds on ten processors (a speedup of 6.2). The equivalent FORTRAN program ran in 54.0 seconds on one processor. Although both phases of a step (finding the pivot element and reducing the matrix) are data independent, neither phase is computationally intensive. In our implementation sequential work accounted for 6% of the execution time. While 6% seems small, it is enough to limit speedup on ten processors to at most 6.4.

4.3 RICARD

RICARD [4] 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. 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.

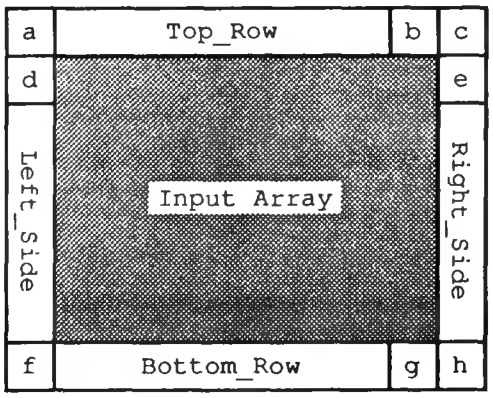
The static SISAL program builds 29 arrays. The memory for all the arrays was preallocated, and 28 of the arrays were built in place. The one array not built in-place was constructed during program initialization, thus the copying was inconsequential. Before optimization, the static program included 17 copy operations and 207 reference count operations. After optimization, there were no unconditional copy operations, 6 conditional copies, 7 reference count operations, and 5 artificial dependency edges. The 6 conditional copy operations were introduced because of *row sharing*. Initially, a number of the arrays share 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. Thus, the conditional copies executed only once each.

For a 1315 level, 5 protein problem, the execution times of the optimized SISAL code were 31.00 hours and 3.45 hours on one and ten processors, respectively (a speedup of 9.0). In comparison, the equivalent FORTRAN program ran in 30.63 hours on one processor. The small differential in sequential times and the near linear speedup are a testimony to the compiler's ability to remove the overhead introduced by applicative semantics.

4.4 SIMPLE

SIMPLE [6] 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 is provided to determine the relation between state variables. The implementation of SIMPLE in SISAL 1.2 is straightforward and exposes considerable parallel work. Two potential sources of copying are the functions `Node_Reflect` and `Zone_Reflect` that build borders around arrays.

`Node_Reflect` takes an *n x m* array as input and returns the array



where a, b, c, d, e, f, g, h are scalars, `Top_Row` and `Bottom_Row` are *(m - 1)*-element row vectors, and `Left_Side` and `Right_Side` are *(n - 1)*-element column vectors. The computations of the border elements are data independent and can be computed in parallel. The most natural way to express the computation in SISAL 1.2 is to compute each element separately and then glue the pieces together as follows (let *X* be the input array),

```
let
   frist_row   := array_addl(array_addh(array_addh(Top_Row, b), c), a);
   second_row  := array_addl(array_addh(X[1], e), d);
   last_row    := array_addl(array_addh(array_addh(Bottom_Row, g), h), f);
   middle_row  := for i in 2, n
                  returns array of
                     array_addl(array_addh(X[i], Right_Side[i-1]),
                                Left_Side[i-1])
                  end for
in
   first_row || second_row || middle_row || last_row
 end let
```

Note that unless the pieces are built in place, each element is copied twice. Although the analysis is complicated, IF2MEM inserts code into the computation graph that computes the size and location of each piece in the final array, and rewires the graph to build the array in place.

The static SISAL program builds 261 arrays. The memory for all the arrays was preallocated, and all the arrays were built in place. Before optimization, the program included 214 copy operations and 2066 reference count operations. After optimization, the program included no unconditional copy operations, 19 conditional copies, 61 reference count operations, and 347 artificial dependency edges. The 19 conditional copy operations were introduced because of row sharing. They executed only once each.

The optimized SISAL code completed 62 iterations for a 100 x 100 grid in 3099.3 seconds and 422.0 seconds on one and ten processors, respectively (a speedup of 7.3). The equivalent FORTRAN programs executed in 3081.3 seconds on one processor. 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 noticed the same phenomenon in some of the Livermore Loops that handled two- and three-dimensional arrays.

4.5 Parallel Simulated Annealing

Simulate 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; 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.

OSC had little difficulty optimizing the computation graph. The static program builds 46 arrays. Memory for all the arrays was preallocated, and all but 4 of the arrays were built in place. The unoptimized static program included 18 copy and 696 reference count operations. The optimized static program included no absolute copy operations, 4 conditional copy operations, 41 reference count operations, and 168 artificial dependency edges. 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 can not be built in place. However, the SISAL runtime 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 updated in place), but its physical size remains constant; i.e., the physical space is not released. Then if an element is added to the high-end of the array, there will be space for the element and we avoid copying (assuming the array can be updated in place). If there is no space for the element, the runtime system allocates a new, larger space and copies the array. When the system allocates new space, it always allocates a few extra bytes to accomodate future growth. In the parallel simulated annealing code, the periods are continually growing and shrinking as tuples are removed and added. Although the compiler does not mark the new periods for build-in-place, the runtime system (except for the first time) always finds room for the new tuples. This implementation of array storage saves over 15000 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 optimized SISAL program ran in 956.2 seconds and 267.8 seconds on one and five processors, respectively (a speedup of 3.6). The speedup is quite good given the fact that the update of the schedule is sequential. The equivalent FORTRAN program executed in 476.6 seconds on one processor, about twice as fast as the SISAL program. The discrepancy in times is due to the allocation and deallocation of the data structures for the move set during every iteration in the SISAL program. 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.

5.0 Conclusions

Table I clearly illustrates why applicative languages have a reputation for inefficiency; however, it also illustrates just as clearly that the SISAL 1.2 compiler can eliminate this inefficiency. The speedups and the comparison of SISAL and FORTRAN execution times on one processor show that, with appropriate optimization, applicative semantics are not a deterrent to high-performance parallel computing on shared-memory multiprocessors. The scientific community should no longer 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 alternative to conventional programming languages on shared-memory multiprocessors. While we acknowledge that the FORTRAN compiler on the Sequent Balance is poor, so is the C compiler used by CC on the Sequent Balance. With respect to conventional optimizations and code generation, we believe that there is as much room for improvement in the FORTRAN compiler as there is in the SISAL and C compilers.

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. Second, to enhance expressive power, we are adding high-order functions, user-defined reductions, infix array operations, subarray operations, parameterized types, and a foreign language interface. We are also unifying 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. Abramson, D. *Using Simulated Annealing to Solve School Timetables: Serial and Parallel Algorithms.* RMIT Technical Report TR-112-069R, Royal Melbourne Institute of Technology, Melbourne, Australia, 1988.

2. Cann, D. C. and R. R. Oldehoeft. *Reference count and copy elimination for parallel applicative computing.* Department of Computer Science Technical Report CS-88-129, Colorado State University, Fort Collins, CO, November 1988.

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

4. 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. *Analytical Biochemistry* **175,** 2(December 1988), pp. 462-473.

5. Cohen, J. Garbage collection of linked data structures. *ACM Computing Surveys* **13,** 3 (September 1981), pp. 341-367.

6. Crowley, W. P., C. P. Hendrickson, and T. E. Rudy. *The SIMPLE Code.* Lawrence Livermore National Laboratory Technical Report UCID-17715, Lawrence Livermore National Laboratory, Livermore, CA, February 1978.

7. Feo, J. T. An analysis of the computational and parallel complexity of the Livermore Loops. *Parallel Computer* **8,** 7 (July 1988), pp. 163-185.

8. Feo, J.T., D. C. Cann and R. R. Oldehoeft. *A report on the SISAL language project.* Lawrence Livermore National Laboratory Technical Report UCRL-10440, Lawrence Livermore National Laboratory, Livermore, CA, January 1990.

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

10. Hudak, P. A semantic model of reference counting and its abstraction. *Proc. of the ACM Conference on Lisp and Functional Programming,* Cambridge, MA, August 1986, pp. 351-363.

11. Hudak, P. et. al. *Report on the Programming Language Haskell, A Non-Strict Purely Functional Language (Version 1.0).* Department of Computer Science Technical Report RR777, Yale University, New Haven, CT, April 1990.

12. Lee, C-C., S. K. Skedzielewski, and J. T. Feo. 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.

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

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

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

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

17. 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.

18. Richert, T.R. *Efficient task management for SISAL.* Department of Computer Science Technical Report 89-111, Colorado State University, Fort Collins, CO, July 1989.

19. Skedzielewski, S. K. and J. Glauert. *IF1 - An intermediate form for applicative languages.* Lawrence Livermore National Laboratory Manual M-170, Lawrence Livermore National Laboratory, Livermore, CA, July 1985.

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

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

22. 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.